GnuCOBOL Programmer's Guide

For Version 3.2 [25 January 2023 at 18:20 GMT.] and for 4.0 (partial & tentative only).

Gary L. Cutler (cutlergl@gmail.com).
For updates Vincent B. Coen (vbcoen@gmail.com).

This manual documents GnuCOBOL 3.2, 25 January 2023 at 18:20 GMT. build.

Document Copyright 2009-2014 Gary L. Cutler, FSF (Free Software Foundation).
Updates: Copyright 2014-2023 Vincent B. Coen, Gary L. Cutler & FSF.

The authors and copyright holders of the COBOL programming language itself used herein:

FLOW-MATIC (trademark for Sperry Rand Corporation) Programming for the Univac(R) I & II. Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM commercial translator form F28-8013, copyrighted 1959 by IBM; FACT DSI27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell, have specifically authorised the use of this material in whole or in part of the COBOL specifications. Such authorisation extends to the reproduction & use of COBOL specifications in programming manuals or similar publications.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License [FDL], Version 1.3 or any later version published by the Free Software Foundation; with Invariant Section "Introduction", no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Table of Contents

GnuCOBOL Programmer's Guide 1

1. Introduction 3

1.1. Additional Reference Sources 3
1.2. Introducing COBOL 4
1.2.1. Why YOU Should Learn COBOL 4
1.2.2. Programmer Productivity 6
1.3. So What is GnuCOBOL? 8

2. COBOL Fundamentals 9

2.1. The COBOL Language - The Basics 9
2.1.1. Language Reserved Words 9
2.1.2. User-Defined Words 10
2.1.3. Case Insensitivity 11
2.1.4. Readability of Programs 11
2.1.5. Divisions Organize Programs 14
2.1.6. Copybooks 14
2.1.7. Structured Data 14
2.1.8. Files 14
2.1.9. Table Handling 19
2.1.10. Sorting and Merging Data 19
2.1.11. String Manipulation 20
2.1.12. Screen Formatting Features 22
2.1.12.1. A Sample Screen 23
2.1.12.2. Color Palette and Video Attributes 23
2.1.13. Report Writer Features 25
2.1.14. Data Initialization 26
2.1.15. Syntax Diagram Conventions 27
2.1.16. Format of Program Source Lines 28
2.1.17. Program Structure 31
2.1.18. Comments 33
2.1.19. Literals 34
2.1.19.1. Numeric Literals 34
2.1.19.2. Alphanumeric Literals 35
2.1.19.3. Figurative Constants 36
2.1.20. Punctuation 37
2.1.21. Interfacing to Other Environments 37
2.2. The COBOL Language - Advanced Techniques 38
2.2.1. Table References 38
2.2.2. Qualification of Data Names 39
2.2.3. Reference Modifiers 40

2.2.4. Arithmetic Expressions ... 42
2.2.5. Conditional Expressions ... 45
2.2.5.1. Condition Names ... 45
2.2.5.2. Class Conditions ... 46
2.2.5.3. Sign Conditions ... 48
2.2.5.4. Switch-Status Conditions ... 49
2.2.5.5. Relation Conditions ... 50
2.2.5.6. Combined Conditions ... 52
2.2.5.7. Negated Conditions ... 53
2.2.6. Use of Periods ... 54
2.2.7. Use of VERB/END-VERB Constructs ... 55
2.2.8. Concurrent Access to Files ... 57
2.2.8.1. File Sharing ... 57
2.2.8.2. Record Locking ... 59

3. CDF - Compiler Directing Facility ... 61
3.1. >>CALL-CONVENTION ... 62
3.2. COPY ... 63
3.3. REPLACE ... 65
3.4. >>DEFINE ... 68
3.5. >>IF ... 69
3.6. >>SET ... 71
3.7. >>SOURCE ... 73
3.8. >>TURN ... 74
3.9. >>D ... 75
3.10. >>DISPLAY ... 76
3.11. >>PAGE ... 77
3.12. >>LISTING ... 78
3.13. >>LEAP-SECONDS ... 79
3.14. $ Directives ... 80
3.15. Predefined compilation variables ... 82

4. IDENTIFICATION DIVISION ... 85

5. ENVIRONMENT DIVISION ... 87
5.1. CONFIGURATION SECTION ... 88
5.1.1. SOURCE-COMPUTER ... 89
5.1.2. OBJECT-COMPUTER ... 90
5.1.3. SPECIAL-NAMES ... 92
5.1.3.1. Alphabet-Name-Clause ... 96
5.1.3.2. Class-Definition-Clause ... 98
5.1.3.3. Switch-Definition-Clause ... 99
5.1.3.4. Symbolic-Characters-Clause ... 100
5.1.4. REPOSITORY ... 101
5.2. INPUT-OUTPUT SECTION ... 103
5.2.1. SELECT ... 104

5.2.1.1. ORGANIZATION SEQUENTIAL 109
5.2.1.2. ORGANIZATION LINE SEQUENTIAL 111
5.2.1.3. ORGANIZATION RELATIVE 113
5.2.1.4. ORGANIZATION INDEXED 115
5.2.2. SAME RECORD AREA 117
5.2.3. MULTIPLE FILE 118

6. DATA DIVISION 119
6.1. Data Definition Principles 120
6.2. FILE SECTION 123
6.2.1. File/Sort-Description 124
6.2.2. FILE-SECTION-Data-Item 127
6.3. WORKING-STORAGE SECTION 129
6.4. LOCAL-STORAGE SECTION 131
6.5. LINKAGE SECTION 133
6.6. REPORT SECTION 135
6.6.1. Report Group Definitions 139
6.6.2. REPORT SECTION Data Items 141
6.7. SCREEN SECTION 143
6.8. Special Data Items 146
6.8.1. 01-Level Constants 146
6.8.2. 66-Level Data Items 150
6.8.3. 77-Level Data Items 151
6.8.4. 78-Level Data Items 152
6.8.5. 88-Level Data Items 153
6.9. Data Description Clauses 154
6.9.1. ANY LENGTH 154
6.9.2. AUTO 155
6.9.3. AUTO-SKIP 156
6.9.4. AUTOTERMINATE 157
6.9.5. BACKGROUND-COLOR 158
6.9.5B. BEFORE TIME 159
6.9.6. BASED 160
6.9.7. BEEP 161
6.9.8. BELL 162
6.9.9. BLANK 163
6.9.10. BLANK WHEN ZERO 164
6.9.11. BLINK 165
6.9.12. COLUMN 166
6.9.13. CONSTANT 168
6.9.14. EMPTY-CHECK 169
6.9.15. ERASE 170
6.9.16. EXTERNAL 171
6.9.17. FALSE 172
6.9.18. FOREGROUND-COLOR 173
6.9.19. FROM 174
6.9.20. FULL 175

6.9.21. GLOBAL ... 176
6.9.22. GROUP INDICATE ... 177
6.9.23. HIGHLIGHT ... 178
6.9.24. JUSTIFIED ... 179
6.9.25. LEFTLINE ... 181
6.9.26. LENGTH-CHECK ... 182
6.9.27. LINE ... 183
6.9.27B. LOWER ... 185
6.9.28. LOWLIGHT ... 186
6.9.29. NEXT GROUP ... 187
6.9.30. NO-ECHO ... 188
6.9.30B. NO UPDATE ... 189
6.9.31. OCCURS ... 190
6.9.32. OVERLINE ... 193
6.9.33. PICTURE ... 194
6.9.34. PRESENT WHEN ... 201
6.9.35. PROMPT ... 202
6.9.36. PROTECTED ... 203
6.9.37. REDEFINES ... 204
6.9.38. RENAMES ... 205
6.9.39. REQUIRED ... 206
6.9.40. REVERSE-VIDEO ... 207
6.9.41. SAME AS ... 208
6.9.41B. SCROLL DOWN ... 210
6.9.41C. SCROLL UP ... 211
6.9.42. SECURE ... 212
6.9.43. SIGN IS ... 213
6.9.43B. SIZE ... 214
6.9.44. SOURCE ... 215
6.9.45. SUM OF ... 216
6.9.46. SYNCRONIZED ... 218
6.9.46B. TIME OUT ... 220
6.9.47. TO ... 221
6.9.48. TYPE ... 222
6.9.49. TYPEDEF ... 227
6.9.50. UNDERLINE ... 229
6.9.50B. UPDATE ... 230
6.9.50C. UPPER ... 231
6.9.51. USAGE ... 232
6.9.52. USING ... 243
6.9.53. VALUE ... 244

7. PROCEDURE DIVISION ... 247

7.1. PROCEDURE DIVISION USING ... 248
7.2. PROCEDURE DIVISION CHAINING ... 250
7.3. PROCEDURE DIVISION RETURNING ... 252
7.4. PROCEDURE DIVISION Sections and Paragraphs ... 253

7.5. DECLARATIVES 254
7.6. Common Clauses on Executable Statements 256
7.6.1. AT END + NOT AT END 256
7.6.2. CORRESPONDING 258
7.6.3. INVALID KEY + NOT INVALID KEY 259
7.6.4. ON EXCEPTION + NOT ON EXCEPTION 260
7.6.5. ON OVERFLOW + NOT ON OVERFLOW 260
7.6.6. ON SIZE ERROR + NOT ON SIZE ERROR 261
7.6.7. ROUNDED 261
7.7. Special Registers 265
7.8. GnuCOBOL Statements 268
7.8.1. ACCEPT 268
7.8.1.1. ACCEPT FROM CONSOLE 268
7.8.1.2. ACCEPT FROM COMMAND-LINE 269
7.8.1.3. ACCEPT FROM ENVIRONMENT 271
7.8.1.4. ACCEPT data-item 272
7.8.1.5. ACCEPT FROM DATE/TIME 279
7.8.1.6. ACCEPT FROM Screen-Info 280
7.8.1.7. ACCEPT FROM Runtime-Info 281
7.8.1.8. ACCEPT OMITTED 282
7.8.1.9. ACCEPT FROM EXCEPTION STATUS 283
7.8.2. ADD 284
7.8.2.1. ADD TO 284
7.8.2.2. ADD GIVING 286
7.8.2.3. ADD CORRESPONDING 288
7.8.3. ALLOCATE 290
7.8.4. ALTER 292
7.8.5. CALL 293
7.8.6. CANCEL 298
7.8.7. CLOSE 299
7.8.8. COMMIT 300
7.8.9. COMPUTE 301
7.8.10. CONTINUE 303
7.8.11. DELETE 304
7.8.12. DISPLAY 305
7.8.12.1. DISPLAY UPON device 305
7.8.12.2. DISPLAY UPON COMMAND-LINE 307
7.8.12.3. DISPLAY UPON ENVIRONMENT-NAME 308
7.8.12.4. DISPLAY Data-Item 309
7.8.12.5. DISPLAY data-item 311
7.8.13. DIVIDE 312
7.8.13.1. DIVIDE INTO 312
7.8.13.2. DIVIDE INTO GIVING 314
7.8.13.3. DIVIDE BY GIVING 316
7.8.14. ENTRY 318
7.8.15. EVALUATE 320
7.8.15B. EXAMINE 324
7.8.16. EXHIBIT 326

7.8.17. EXIT ... 328
7.8.18. FREE ... 331
7.8.19. GENERATE ... 332
7.8.20. GOBACK ... 334
7.8.21. GO TO ... 335
7.8.21.1. Simple GO TO ... 335
7.8.21.2. GO TO DEPENDING ON ... 336
7.8.22. IF ... 338
7.8.23. INITIALIZE ... 339
7.8.24. INITIATE ... 343
7.8.25. INSPECT ... 344
7.8.26. MERGE ... 349
7.8.27. MOVE ... 352
7.8.27.1. Simple MOVE ... 352
7.8.27.2. MOVE CORRESPONDING ... 353
7.8.28. MULTIPLY ... 354
7.8.28.1. MULTIPLY BY ... 354
7.8.28.2. MULTIPLY GIVING ... 356
7.8.29. OPEN ... 358
7.8.30. PERFORM ... 360
7.8.30.1. Procedural PERFORM ... 360
7.8.30.2. Inline PERFORM ... 362
7.8.30.3. VARYING ... 363
7.8.31. READ ... 366
7.8.31.1. Sequential READ ... 366
7.8.31.2. Random READ ... 368
7.8.32. READY TRACE ... 370
7.8.33. RELEASE ... 371
7.8.34. RESET TRACE ... 372
7.8.35. RETURN ... 373
7.8.36. REWRITE ... 374
7.8.37. ROLLBACK ... 376
7.8.38. SEARCH ... 377
7.8.39. SEARCH ALL ... 379
7.8.40. SET ... 382
7.8.40.1. SET ENVIRONMENT ... 382
7.8.40.2. SET Program-Pointer ... 383
7.8.40.3. SET ADDRESS ... 384
7.8.40.4. SET Index ... 385
7.8.40.5. SET UP/DOWN ... 386
7.8.40.6. SET Condition Name ... 387
7.8.40.7. SET Switch ... 388
7.8.40.8. SET ATTRIBUTE ... 389
7.8.40.9. SET LAST EXCEPTION ... 390
7.8.41. SORT ... 391
7.8.41.1. File-Based SORT ... 391
7.8.41.2. Table SORT ... 395
7.8.42. START ... 397

7.8.43. STOP ... 399
7.8.44. STRING ... 401
7.8.45. SUBTRACT ... 403
7.8.45.1. SUBTRACT FROM ... 403
7.8.45.2. SUBTRACT GIVING ... 405
7.8.45.3. SUBTRACT CORRESPONDING ... 407
7.8.46. SUPPRESS ... 409
7.8.47. TERMINATE ... 410
7.8.48. TRANSFORM ... 411
7.8.49. UNLOCK ... 412
7.8.50. UNSTRING ... 413
7.8.51. WRITE ... 417

8. Functions ... 421
8.1. Intrinsic Functions ... 421
8.1.1. ABS ... 426
8.1.2. ACOS ... 427
8.1.3. ANNUITY ... 428
8.1.4. ASIN ... 429
8.1.5. ATAN ... 430
8.1.6. BIT-OF ... 431
8.1.7. BIT-TO-CHAR ... 432
8.1.8. BYTE-LENGTH ... 433
8.1.9. CHAR ... 434
8.1.10. COMBINED-DATETIME ... 435
8.1.11. CONCAT ... 436
8.1.11. CONCATENATE ... 437
8.1.12. CONTENT-LENGTH ... 438
8.1.13. CONTENT-OF ... 439
8.1.14. COS ... 440
8.1.15. CURRENCY-SYMBOL ... 441
8.1.16. CURRENT-DATE ... 442
8.1.17. DATE-OF-INTEGER ... 443
8.1.18. DATE-TO-YYYYMMDD ... 444
8.1.19. DAY-OF-INTEGER ... 445
8.1.20. DAY-TO-YYYYDDD ... 446
8.1.21. E ... 447
8.1.22. EXCEPTION-FILE ... 448
8.1.23. EXCEPTION-LOCATION ... 449
8.1.24. EXCEPTION-STATEMENT ... 450
8.1.25. EXCEPTION-STATUS ... 451
8.1.26. EXP ... 453
8.1.27. EXP10 ... 454
8.1.28. FACTORIAL ... 455
8.1.29. FORMATTED-CURRENT-DATE ... 456
8.1.30. FORMATTED-DATE ... 457
8.1.31. FORMATTED-DATETIME ... 458

8.1.32. FORMATTED-TIME ... 459
8.1.33. FRACTION-PART ... 460
8.1.34. HEX-OF ... 461
8.1.35. HEX-TO-CHAR ... 462
8.1.36. HIGHEST-ALGEBRAIC ... 463
8.1.37. INTEGER ... 464
8.1.38. INTEGER-OF-DATE ... 465
8.1.39. INTEGER-OF-DAY ... 466
8.1.40. INTEGER-OF-FORMATTED-DATE ... 467
8.1.41. INTEGER-PART ... 468
8.1.42. LENGTH ... 469
8.1.43. LENGTH-AN ... 470
8.1.44. LOCALE-COMPARE ... 471
8.1.45. LOCALE-DATE ... 472
8.1.46. LOCALE-TIME ... 473
8.1.47. LOCALE-TIME-FROM-SECONDS ... 474
8.1.48. LOG ... 475
8.1.49. LOG10 ... 476
8.1.50. LOWER-CASE ... 477
8.1.51. LOWEST-ALGEBRAIC ... 478
8.1.52. MAX ... 479
8.1.53. MEAN ... 480
8.1.54. MEDIAN ... 481
8.1.55. MIDRANGE ... 482
8.1.56. MIN ... 483
8.1.57. MOD ... 484
8.1.58. MODULE-CALLER-ID ... 485
8.1.59. MODULE-DATE ... 486
8.1.60. MODULE-FORMATTED-DATE ... 487
8.1.61. MODULE-ID ... 488
8.1.62. MODULE-PATH ... 489
8.1.63. MODULE-SOURCE ... 490
8.1.64. MODULE-TIME ... 491
8.1.65. MONETARY-DECIMAL-POINT ... 492
8.1.66. MONETARY-THOUSANDS-SEPARATOR ... 493
8.1.67. NUMERIC-DECIMAL-POINT ... 494
8.1.68. NUMERIC-THOUSANDS-SEPARATOR ... 495
8.1.69. NUMVAL ... 496
8.1.70. NUMVAL-C ... 497
8.1.70B. NUMVAL-C-V2 ... 499
8.1.71. NUMVAL-F ... 500
8.1.72. ORD ... 501
8.1.73. ORD-MAX ... 502
8.1.74. ORD-MIN ... 503
8.1.75. PI ... 504
8.1.76. PRESENT-VALUE ... 505
8.1.77. RANDOM ... 506
8.1.78. RANGE ... 508

8.1.79. REM ... 509
8.1.80. REVERSE ... 510
8.1.81. SECONDS-FROM-FORMATTED-TIME ... 511
8.1.82. SECONDS-PAST-MIDNIGHT ... 512
8.1.83. SIGN ... 513
8.1.84. SIN ... 514
8.1.85. SQRT ... 515
8.1.86. STANDARD-DEVIATION ... 516
8.1.87. STORED-CHAR-LENGTH ... 517
8.1.88. SUBSTITUTE ... 518
8.1.89. SUBSTITUTE-CASE ... 519
8.1.90. SUM ... 520
8.1.91. TAN ... 521
8.1.92. TEST-DATE-YYYYMMDD ... 522
8.1.93. TEST-DAY-YYYYDDD ... 523
8.1.94. TEST-FORMATTED-DATETIME ... 524
8.1.95. TEST-NUMVAL ... 525
8.1.96. TEST-NUMVAL-C ... 526
8.1.97. TEST-NUMVAL-F ... 527
8.1.98. TRIM ... 528
8.1.99. UPPER-CASE ... 529
8.1.100. VARIANCE ... 530
8.1.101. WHEN-COMPILED ... 531
8.1.102. YEAR-TO-YYYY ... 532
8.1.103. BOOLEAN-OF-INTEGER ... 533
8.1.104. CHAR-NATIONAL ... 534
8.1.105. DISPLAY-OF ... 535
8.1.106. EXCEPTION-FILE-N ... 536
8.1.107. EXCEPTION-LOCATION-N ... 537
8.1.108. INTEGER-OF-BOOLEAN ... 538
8.1.109. NATIONAL-OF ... 539
8.1.110. STANDARD-COMPARE ... 540
8.2. Built-In System Subroutines ... 542
8.2.1. C$CALLEDBY ... 544
8.2.2. C$CHDIR ... 545
8.2.3. C$COPY ... 546
8.2.4. C$DELETE ... 547
8.2.5. C$FILEINFO ... 548
8.2.6. C$GETPID ... 549
8.2.7. C$JUSTIFY ... 550
8.2.8. C$MAKEDIR ... 551
8.2.9. C$NARG ... 552
8.2.10. C$PARAMSIZE ... 553
8.2.11. C$PRINTABLE ... 554
8.2.12. C$SLEEP ... 555
8.2.13. C$TOLOWER ... 556
8.2.14. C$TOUPPER ... 557
8.2.15. CBL_AND ... 558

8.2.16. CBL_CHANGE_DIR ... 559
8.2.17. CBL_CHECK_FILE_EXIST ... 560
8.2.18. CBL_CLOSE_FILE ... 561
8.2.19. CBL_COPY_FILE ... 562
8.2.20. CBL_CREATE_DIR ... 563
8.2.21. CBL_CREATE_FILE ... 564
8.2.22. CBL_DELETE_DIR ... 565
8.2.23. CBL_DELETE_FILE ... 566
8.2.24. CBL_EQ ... 567
8.2.25. CBL_ERROR_PROC ... 568
8.2.26. CBL_EXIT_PROC ... 570
8.2.27. CBL_FLUSH_FILE ... 572
8.2.28. CBL_GC_FORK ... 573
8.2.29. CBL_GC_GETOPT ... 575
8.2.30. CBL_GC_HOSTED ... 577
8.2.31. CBL_GC_NANOSLEEP ... 580
8.2.32. CBL_GC_PRINTABLE ... 581
8.2.33. CBL_GC_WAITPID ... 582
8.2.34. CBL_GET_CSR_POS ... 583
8.2.35. CBL_GET_CURRENT_DIR ... 584
8.2.36. CBL_GET_SCR_SIZE ... 585
8.2.37. CBL_IMP ... 586
8.2.38. CBL_NIMP ... 587
8.2.39. CBL_NOR ... 588
8.2.40. CBL_NOT ... 589
8.2.42. CBL_OPEN_FILE ... 590
8.2.43. CBL_OR ... 591
8.2.44. CBL_READ_FILE ... 592
8.2.45. CBL_READ_KBD_CHAR ... 593
8.2.46. CBL_RENAME_FILE ... 594
8.2.47. CBL_SET_CSR_POS ... 595
8.2.48. CBL_TOLOWER ... 596
8.2.49. CBL_TOUPPER ... 596
8.2.50. CBL_WRITE_FILE ... 597
8.2.51. CBL_XOR ... 598
8.2.52. SYSTEM ... 599
8.2.53. X"91" ... 600
8.2.54. X"E4" ... 601
8.2.55. X"E5" ... 601
8.2.56. X"F4" ... 602
8.2.57. X"F5" ... 603

9. Report Writer Usage ... 605

9.1. RWCS Lexicon ... 605
9.2. The Anatomy of a Report ... 606
9.3. The Anatomy of a Report Page ... 607
9.4. How RWCS Builds Report Pages ... 608

9.5. Control Hierarchy ... 609
9.6. An Example ... 611
9.6.1. Data ... 611
9.6.2. Program ... 613
9.6.3. Generated Report Pages ... 618
9.7. Control Hierarchy (Revisited) ... 624
9.8. Turning PHYSICAL Page Formatting Into LOGICAL Formatting ... 627

10. Interfacing With The OS ... 629

10.1. Compiling Programs ... 629
10.1.1. cobc - The GnuCOBOL Compiler - Runtime options ... 629
10.1.1.1. cobc option -Xref an example ... 641
10.1.1.2. Cross Reference listing using cobxref ... 643
10.1.2. Compilation Time Environment Variables ... 645
10.1.3. Predefined Compilation Variables ... 646
10.1.4. Locating Copybooks ... 648
10.1.5. Compiler Configuration Files ... 649
10.2. Running Programs ... 656
10.2.1. Direct Execution ... 656
10.2.2. Executing Dynamically-Loadable Libraries ... 656
10.2.2.1. cobcrun - Command-line Execution ... 656
10.2.2.2. Dynamically Loaded Subprograms ... 657
10.2.3. Run Time Environment Variables ... 658
10.2.3.1. General instructions ... 660
10.2.3.2. General Environment ... 661
10.2.3.3. Call Environment ... 664
10.2.3.4. File I/O ... 665
10.2.3.5. Screen I/O ... 670
10.2.3.6. Report I/O ... 673
10.2.3.7. File I/O Environment Variables and/or dictionary file ... 673
10.2.4. Program Arguments ... 675
10.3. Binary Truncation ... 675

11. Sub-Programming ... 679

11.1. Subprogram Types ... 679
11.2. Independent vs Contained vs Nested Subprograms ... 679
11.3. Alternate Entry Points ... 681
11.4. Dynamic vs Static Subprograms ... 681
11.5. Subprogram Execution Flow ... 683
11.5.1. Subroutine Execution Flow ... 683
11.5.2. User-Defined Function Execution Flow ... 684
11.6. Sharing Data Between Calling and Called Programs ... 686
11.6.1. Subprogram Arguments ... 686
11.6.1.1. Calling Program Considerations ... 686
11.6.1.2. Called Program Considerations ... 687
11.6.2. GLOBAL Data Items ... 687
11.6.3. EXTERNAL Data Items ... 688

11.7. Recursive Subprograms 689
11.8. Combining GnuCOBOL and C Programs 692
11.8.1. GnuCOBOL Run-Time Library Requirements 692
11.8.2. String Allocation Differences Between GnuCOBOL and C .. 692
11.8.3. Matching C Data Types with GnuCOBOL USAGE's 693
11.8.4. GnuCOBOL Main Programs CALLing C Subprograms .. 694
11.8.5. C Main Programs Calling GnuCOBOL Subprograms 696

12. Programming Style Suggestions 699
12.1. Marking Changes in Programs 699
12.2. Data Item Coding and Naming Conventions 700
12.3. Table Subscripting versus Table Indexing 702
12.4. Copybook Naming Conventions and Usage 703
12.5. PROCEDURE DIVISION Sections Versus Paragraphs 704
12.6. COMPUTE Versus ADD-SUBTRACT-MULTIPLY-DIVIDE .. 705

13. Programming for XFD 707
13.1. GnuCobol use SQL for files 707

Appendix A - Glossary of Terms 719

Appendix B - Reserved Word Lists 727

Appendix C1 - Grouped Word Lists by feature and function 729
Common reserved words 729
Extra (obsolete) context sensitive words 750
Internal registers 750

Appendix C2 - Intrinsic Functions 753

Appendix C3 - System routines 757

Appendix C4 - System names 759
System names: device 759
System names: feature 759
System names: switch 759

Appendix C5 - Exceptions 761

Appendix D - GNU Free Documentation License .. 767

Appendix E - Summary of Document Changes . . 775

Index 779

GnuCOBOL Programmer's Guide

This manual documents the 25 January 2023 at 18:20 GMT. build of GnuCOBOL 3.2.

1. Introduction

This document describes the syntax, semantics and use of the COBOL programming language as implemented by GnuCOBOL, formerly known as OpenCOBOL.

The original principal developers of GnuCOBOL were Keisuke Nishida and Roger While. Since then, many members of the community have been involved in its development.

This document is intended to serve as a fully functional reference and light user's guide, suitable for both those readers learning COBOL for the first time as a light training tool, as well as those already familiar with another dialect of COBOL.

A separate manual — containing only the basic details of the GnuCOBOL implementation and designed for experienced COBOL programmers — has been taken from this guide. That document (*GnuCOBOL Quick Reference*) contains *no* training subject matter.

A new manual that is based on this manual (Programmers Guide) is the Programmers Reference. This will have any training material removed so it will be a detailed Reference only. There are a few issues to overcome in that it uses currently the same text for the Guide and this will need changing in some areas in order to remove any training materials creating new text documents used as input to build this manual.

Other documents that should be read is the `gnucobol.pdf` found in the `doc` directory of the compiler sources and the file `NEWS` supplied with the source code of the GnuCOBOL compiler, in the top-level directory. There you will find the latest COBOL language features that have been added, some of which may not be in this document due to time constraints. If you find any, please report it as a bug for the Programmer's Guide so that it can be fixed.

Yet another document which delves deeper in to the compiler that is a *must* read, is the FAQ available via the GnuCOBOL Manuals and Guides (`https://gnucobol.sourceforge.io/#faq`), although it could do with a wee clean up to ease reading and finding required information.

1.1. Additional Reference Sources

For those wishing to learn COBOL for the first time, Gary can strongly recommend the following resources.

If you like to hold a book in your hands, I strongly recommend *Murach's Structured COBOL*, by Mike Murach, Anne Prince and Raul Menendez (2000) - ISBN 9781890774059. Mike Murach and his various writing partners have been writing outstanding COBOL textbooks for decades. It's an excellent book for those familiar with the concepts of programming in other languages, but unfamiliar with COBOL.

Another to consider is COBOL Unleashed by Jon Wessler, et al (with many other contributors) dated 1998 - ISBN 9780672312540 - 1,000 pages, which is perhaps more geared to the mainframe but with much related to other platform including for GnuCOBOL which Vincent finds very useful at times.

One more is Beginning COBOL for Programmers Paperback 20 Mar. 2014 by Michael Coughlan - ISBN 978-1430262534 which may well be worth a read. There are many others.

Would you prefer a web-based tutorial? Try the University of Limerick (Ireland) - COBOL web site (`http://www.csis.ul.ie/cobol/`).

In addition there is the GnuCOBOL FAQ (`https://gnucobol.sourceforge.io/faq/index.html`) — which has now exceeded 1,400 pages — available as HTML or a downloadable `.pdf` file.

Along with every release of the compiler sources is the file `NEWS`. It contains up to the minute updates regarding the compiler and additional COBOL language elements which may well *not* be yet included in this manual.

1.2. Introducing COBOL

If you already know a programming language other than COBOL, chances are that language is Java, C or C++. You will find COBOL much different from those; sometimes the differences are a good thing and sometimes they aren't. The thing to remember about COBOL is this: *it was designed to solve business problems.*

COBOL, first introduced to the programming public in 1959, was the very first programming language to become standardized (in 1960). This meant that a standard-compliant COBOL program written on computer "A" made by company "B" would be able to be compiled and executed on computer "X" made by company "Y" with very few, if any, changes. This may not seem like such a big deal today, but it was a radical departure from all programming languages that came before it and even many that came after it.

The name COBOL actually says it all — COBOL is an acronym that stands for "(**CO**)mmon (**B**)usiness (**O**)riented (**L**)anguage". Note the fact that the word "common" comes before all others. The word "business" is a close second. Therein lies the key to COBOL's success.

1.2.1. Why YOU Should Learn COBOL

Despite statements from industry "insiders", the COBOL programming language is not dead, even though newer and so-called "modern" languages like Java, C#, .NET, Ruby on Rails and so on appear to have become the languages of choice in the Information Technology world. These languages have become popular because they address the following desired requirements for "modern" programming:

1. They conform to the principles of Object-Oriented Programming (OOP). This is desired for one major reason — it facilitates "code re-usability", thus improving the productivity of programmers by allowing them to re-use previously written (and debugged) code in new applications. For one reason or another, COBOL is perceived as being weak in this regard. It isn't (especially today), as we'll see in the next section, but perception is important.
2. Those languages aren't limited to mainframe computers, as COBOL is *perceived* to be. Some, like .NET and Ruby, aren't even *available* on mainframes. The "modern" programming languages were designed and intended for use on the full variety of computer platforms, from shirt-pocket computers (i.e. smart phones) up to the most massive of supercomputers.
3. There are several excellent commercially available COBOL implementations available for non-mainframe systems (Micro Focus Visual COBOL, AccuCOBOL, NetCOBOL

and Elastic COBOL, just to name a few), including Windows and UNIX/Linux systems. These aren't cheap, however.

4. Universities love the "Modern" languages. In the U.S., 73% of colleges lack even one COBOL course on their curricula. COBOL, it appears, is no longer "cool" enough for students to fill a classroom.

Just because COBOL doesn't traditionally support objects, classes, and the like doesn't mean that its "procedural" approach to computing isn't valuable — after all, it runs 70% of the *worlds* business transactions, and does so utilising over 200 billion lines of code (according to research):

- Using programs that, for the most part, are much more self-documenting than would be the case with any other programming language.
- Effortlessly providing arithmetic accuracy to 31 digits, with performance approaching that of well-written assembly-language programs. Don't think this isn't critically important to banks, investment houses and any business interested in tracking revenues, expenses and profits (duh - like ALL of them).
- Integrating well with non-COBOL infrastructures such as XML, SOA, MQ, almost any DBMS, Transaction Processing platforms, Queue-Management facilities and other programming languages.
- By running on almost as many different computing platforms as Java can. You can't run COBOL programs in your smart phone, but desktops, workstations, midframes/servers, mainframes and supercomputers are all fair game.

Today's IT managers and business leaders are faced with a challenging dilemma — how do you maintain the enormous COBOL code base that is still running their businesses when academia has all but abandoned the language they need their people to use to keep the wheels rolling? The problem is compounded by the fact that those programmers that are skilled in COBOL are retiring and taking their knowledge with them. In some markets, this appears to be having an inflationary effect on the cost of resources (COBOL programmers) whose supply is becoming smaller and smaller. The pressure to update applications to make use of more up-to-date graphical user interfaces is also perceived as a reason to abandon COBOL in favour of GUI-friendly languages such as Java.

Businesses are addressing the COBOL challenge in different ways:

1. By undertaking so-called "modernization projects", where existing applications are either rewritten in "modern" languages or replaced outright with purchased packages. Most of these businesses are using such activities as an excuse to abandon "expensive" mainframes in favour of (presumably) less-expensive "open systems" (mid frame/server) solutions.
2. Many times these businesses are finding the cost of the system/networking engineering, operational management and monitoring and risk management (i.e. disaster recovery) infrastructures necessary to support truly mission-critical applications to be so high that the "less-expensive" solution really isn't; in these cases the mainframe may remain the best option, thus leaving COBOL in play and businesses seeking another solution for at least part of their application base.

3. Training their own COBOL programmers. Since colleges, universities and technical schools have lost interest in doing so, many businesses have undertaken the task of "growing their own" new crop of COBOL programmers. Fear of being pigeon-holed into a niche technology is a factor inhibiting many of today's programmers from willingly volunteering for such training.
4. By moving the user-interface onto the desktop; such efforts involve running modern-language front-end clients on user desktops (or laptops or smart phones, etc.) with COBOL programs providing server functionality on mainframe or midframe platforms, providing all the database and file "heavy lifting" on the back-end. Solutions like this provide users with the user-interfaces they want/need while still leveraging COBOL's strengths on (possibly) downsized legacy mainframe or midframe systems.

It's probably a true that an IT professional can no longer afford to allow COBOL to be the *only* wrench in their toolbox, but with a massive code base still in production now and for the foreseeable future, adding COBOL to a multi-lingual curriculum vitae (CV) and/or resume (yes — they ARE different) is not a bad thing at all. Knowing COBOL as well as the language du-jour will make you the smartest person in the room when the discussion of migrating the current "legacy" environment to a "modern" implementation comes around.

You'll find COBOL an easy language to learn and a FAR EASIER language to master than many of the "modern" languages.

The whole reason you're reading this is that you've discovered GnuCOBOL — another implementation of COBOL in addition to those mentioned earlier. The distinguishing characteristic of GnuCOBOL versus those others is that GnuCOBOL is FREE open-source and therefore FREE to obtain and use. It is community-enhanced and community-supported. Later in this document (see [So What is GnuCOBOL?], page 8), you'll begin to learn more about this COBOL implementation's capabilities.

1.2.2. Programmer Productivity

Throughout the history of computer programming, the search for new ways to improve of the productivity of programmers has been a major consideration. Other than hobbyists, programming is an activity performed for money, and businesses abhor spending anything more than is absolutely necessary; even government agencies try to spend as little money on projects as is absolutely necessary.

The amount of programming necessary to accomplish a given task — including rework needed by any errors found during testing (testing is sometimes jokingly defined as: *that time during which an application is actually in production, allowing users to discover the problems*) is the measure of programmer productivity. Anything that reduces that effort will therefore reduce the time spent in such activities therefore reducing the expense of same. When the expense of programming is reduced, programmer productivity is increased.

Sometimes the quest for improved programmer productivity (and therefore reduced programming *expense*) has taken the form of introducing new features in programming languages, or even new languages altogether. Sometimes it has resulted in new ways of using the existing languages.

While many technological and procedural developments have made *evolutionary* improvements to programmer productivity, each of the following three events has been responsible for *revolutionary* improvements:

- The development of so-called "higher-level" programming languages that enable a programmer to specify in a single statement of the language an action that would have required many more separate statements in a prior programming language. The standardization of such languages, making them usable on a wide variety of computers and operating systems, was a key aspect of this development. COBOL was a pioneering development in this area, being a direct descendant of the very first higher-level language (FLOW-MATIC, developed by US Naval Lieutenant Grace Hopper) and the first to become standardized.
- The establishment of programming techniques that make programs easier to read and therefore easier to understand. Not only do such techniques reduce the amount of rework necessary simply to make a program work as designed, but they also reduce the amount of time a programmer needs to study an existing program in order how to best adapt it to changing business requirements. The foremost development in this area was structured programming. Introduced in the late 1970's, this approach to programming spawned new programming languages (PASCAL, ALGOL, PL/1 and so forth) designed around it. With the ANSI 85 standard, COBOL embraced the principles espoused by structured programming mavens as well as any of the languages designed strictly around it.
- The establishment of programming techniques AND the introduction of programming language capabilities to facilitate the re-usability of program code. Anything that supports code re-usability can have a profound impact to the amount of time it takes to develop new applications or to make significant changes to existing ones. In recent years, object-oriented programming (OOP) has been the industry "poster child" for code re-usability. By enabling program logic and the data structures that logic manipulates to be encapsulated into easily stored and retrieved (and therefore "reusable") modules called classes, the object-oriented languages such as Java, C++ and C# have become the favourites of academia. Since students are being trained in these languages and only these, by and large, it's no surprise that — today — object-oriented programming languages are the darlings of the industry.

  The reality is, however, that good programmers have been practising code re-usability for more than a half-century. Up until recently, COBOL programmers have had some of the best code re-usability tools available — they've been doing it with copybooks and subprograms rather than classes, methods and attributes but the net results have been similar. With the COBOL2002 and the COBOL 2014 standards, the COBOL programming language has become just as "object-oriented" as the "modern" languages, while preserving the ability to support, modify, compile and execute "legacy" COBOL programs as well.

While GnuCOBOL supports few of the OOP programming constructs defined by the COBOL2002 and COBOL2014 standards, it supports every aspect of the ANSI 85 standard and therefore fully meets the needs of points #1 and #2, above. With its

supported feature set (see [So What is GnuCOBOL?], page 8), it provides significant programmer productivity capabilities.

1.3. So What is GnuCOBOL?

GnuCOBOL is a free and open sourced COBOL compiler and runtime environment, written using the C programming language which itself is free to use and can be used in all forms including for commercial purposes - there is no restrictions. GnuCOBOL is typically distributed in source-code form, and must then be built for your computer's operating system using the system's C compiler and loader. While originally developed for the UNIX and Linux operating systems, GnuCOBOL has also been successfully built for computers running OSX (a OpenBSD implementation) and Windows utilizing the UNIX-emulation features of such tools as Cygwin and MinGW. Also see the GNU website (`https://savannah.gnu.org/projects/gnucobol`) for more information. Pre built Implemeations for Windows can also be found at `https://www.arnoldtrembley.com/GnuCOBOL.htm`.

The MinGW approach is a personal favourite with the author of this manual because it creates a GnuCOBOL compiler and runtime library that require only a single MinGW DLL to be available for the GnuCOBOL compiler, runtime library and user programs. That DLL is freely distributable under the terms of the GNU General Public License. A MinGW build of GnuCOBOL fits easily on and runs from a 128MB flash drive with no need to install any software onto the Windows computer that will be using it. Some functionality of the language, dealing with the sharing of files between concurrently executing GnuCOBOL programs and record locking on certain types of files, is sacrificed however as the underlying operating system routines needed to implement them aren't available to Windows and aren't provided by MinGW. The current version for MinGW is available at the download link along with various other platforms at the GnuCOBOL `https://sourceforge.net/projects/gnucobol/files/gnu-cobol`, download website.

GnuCOBOL has also been built as a truly native Windows application utilizing Microsoft's freely-downloadable Visual Studio Community package to provide the C compiler and linker/loader. This approach does not lend itself well to a "portable" distribution.

The GnuCOBOL compiler generates C code from your COBOL source programs; that C code is then automatically compiled and linked using your system's C compiler (typically, but not limited to, `gcc`).

GnuCOBOL fully supports much of the ANSI 85 standard for COBOL (the only major exclusion is the Communications Module) and also supports some of the components of the COBOL2002 and COBOL2014 standards, such as the `SCREEN SECTION` (see [SCREEN SECTION], page 143), table-based `SORT` (see [Table SORT], page 395) and user-defined functions. There are others with more being added almost weekly.

End of Chapter 1 — Introduction

2. COBOL Fundamentals

This chapter describes the syntax, semantics and usage of the COBOL programming language as implemented by the current version of GnuCOBOL. For the rest of this document the Language is spelt as COBOL to ease reading however the compiler name retains the mixed case of GnuCOBOL.

This document is intended to serve as a full-function reference and user's guide suitable for both those readers learning COBOL for the first time as usage as a training tool, as well as those already familiar with some dialects of the COBOL language.

A separate manual exists that just contains the details of the Cobol grammar as implemented in GnuCOBOL, which is designed strictly for experienced COBOL programmers and this is taken from this guide. This does NOT contain any training subject matter what so ever.

These extra manuals are: GnuCOBOL Quick Reference containing just the COBOL semantics / grammar in a short document while the other, GnuCOBOL Sample Programs, shows detailed example Cobol programs with indication of syntax used in each program.

For each implementation of the GnuCOBOL compiler the supplied files NEWS should also be read for any last minute updates along with files README and INSTALL for building the compiler.

2.1. The COBOL Language - The Basics

2.1.1. Language Reserved Words

Cobol programs consist of a sequence of words and symbols. Words, which consist of sequences of letters (upper- and/or lower-case), digits, dashes ('-') and/or underscores ('_') may have a pre-defined, specific, meaning to the compiler or may be invented by the programmer for his/her purposes.

The GnuCOBOL language specification defines well over 1130 *Reserved Words* — words to which the compiler assigns a special meaning. This list and number applies to the default list which covers many implementations. It is possible to limit the list to either a specific implementation via -std=xyz[-strict] or to manually unreserve words if they are used in existing sources as user-defined words.

Programmers may use a reserved word as *part* of a word they are creating themselves, but may not create their own word as an exact duplicate (without regard to case) of a COBOL reserved word. Note that a reserved word includes all classes, such as intrinsic functions, mnemonics names, system routines. The list of reserved words can be changed by adding or removing specific words for a given compile or as a default by use of the steering command -std= (dialect) and –conf= (users config file). See the specific config files that are by default, held in `/usr/local/share/gnucobol/config`. Also using the option '`FUNCTION ALL INTRINSIC`', will add another 100+ reserved words. These can be modified to match the requirements of a business or project team but be Warned, that these are updated when a new version of the compiler is built so might be more prudent to create your own configuation based on an existing one but with a different name.

In addition, you can add and/or remove reserved words by adding one of these options to cobc to add `-freserved=word` or, to remove, `-fnot-reserved=word`. As well as `-freserved=word:alias` to create an alias for a word as well as `-fnot-register=word` or `-fregister=word` to remove or add, a special register word.

See Appendix's B for a complete list of GnuCOBOL reserved words and C1 - 4 (for grouped word lists).

For any given version of GnuCOBOL you can also list the full current set of reserved words by running cobc with `--list-reserved`, `--list-intrinsic`, `--list-system` as well as `--list-mnemonics`. Again subject to variation depending on usage of the `--std` line command.

2.1.2. User-Defined Words

When you write GnuCOBOL programs, you'll need to create a variety of words to represent various aspects of the program, the program's data and the external environment in which the program will run. This will include internal names by which data files will be referenced, data item names and names of executable logic procedures as section and paragraph names.

User-defined words may be composed from the characters '`A`' through '`Z`' (upper- and/or lower-case), '`0`' through '`9`', dash ('`-`') and underscore ('`_`'). User-defined words may neither start nor end with hyphen or underscore characters.

Other programming languages provide the programmer with a similar capability of creating their own words (names) for parts of a program; COBOL is somewhat unusual when compared to other languages in that user-defined words may *start* with a digit.

With the exception of logic procedure names, which may consist entirely of nothing but digits, user-defined words must contain at least one letter.

The maximum size of a user defined word in Cobol is 31 characters as per the COBOL 2014 Standard but to help support other compilers it can be extended by the usage of -std (COBOL85 and ibm-strict has 30) to increase the limit to 63 characters. It must be pointed out that exceeding the standard limit will seriously restrict the ability of transferring any code written for GnuCOBOL to another brand of compiler without changing all such user defined words to 30 or 31. The whole art of writing using Cobol is to minimise the need to change any code over the years that your programs will be in use.

There are very many examples of programs written going back to the 1960's that are still in operation around the world and the number of lines of Cobol code is estimated at 200 billion.

For example, this author (Vincent Coen) has code going back to the early 60's with admittedly changes over the years, still in full operation, just take a look at the Contrib area and check out cobxref - Cobol Cross Reference listing tool (also on Sourceforge), dectrans - (Decision Translator), flightlog - Pilots Log Book (also on Sourceforge), and also in Sourceforge - ACAS (Applewood Computers Accounting System) - this one with the original code only, going back to 1967.

Of course many if not most of these applications have had many changes, upgrades etc, over the years, but it shows just how long programs written in Cobol have survived and gone on in full time use for some 60 years.

The point is that, when writing in Cobol, you should always consider is, will the code be transferrable to another system or compiler in the years to come, but without going over the top !

2.1.3. Case Insensitivity

All COBOL implementations allow the use of both upper and lower case letters in program coding. GnuCOBOL is completely insensitive to the case used when writing reserved words or user-defined names. Thus, `AAAAA`, `aaaaa`, `Aaaaa` and `AaAaA` are all the same word as far as GnuCOBOL is concerned.

The only time the case used does matter is within quoted character strings, where character values will be exactly as coded.

By convention throughout this document, COBOL reserved words will be shown entirely in `UPPER-CASE` while those words that were created by a programmer will be represented by tokens in mixed or lower case.

This isn't a bad practice to use in actual programs, as it leads to programs where it is much easier to distinguish reserved words from user-defined ones!

2.1.4. Readability of Programs

Critics of COBOL frequently focus on the wordiness of the language, often citing the case of a so-called "Hello World" program as the "proof" that COBOL is so much more tedious to program in than more "modern" languages. This tedium is cited as such a significant impact to programmer productivity that, in their opinions, COBOL can't go away quickly enough.

Here are two different "Hello World" applications, one written in Java and the second in GnuCOBOL. First, the Java version:

```
Class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

And here is the same program, written in GnuCOBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HelloWorld.
PROCEDURE DIVISION.
    DISPLAY "Hello World!".
```

Both of the above programs could have been written on a single line, if desired, and both languages allow a programmer to use (or not use) indentation as they see fit to improve program readability. Sounds like a tie so far.

Let's look at how much more "wordy" COBOL is than Java. Count the characters in the two programs. The Java program has 95 (not counting carriage returns and any indentation). The COBOL program has 89 (again, not counting carriage returns and indentation)! Technically, it could have been only 65 because the `IDENTIFICATION DIVISION.` header is

actually optional. Clearly, "Hello World" doesn't look any more concise in Java than it does in COBOL.

Let's look at a different problem. Surely a program that asks a user to input a positive integer, generates the sum of all positive integers from 1 to that number and then prints the result will be MUCH shorter and MUCH easier to understand when coded in Java than in COBOL, right?

You can be the judge. First, the Java version:

```
import java.util.Scanner;
public class sumofintegers {
    public static void main(String[] arg) {
        System.out.println("Enter a positive integer");
        Scanner scan=new Scanner(System.in);
        int n=scan.nextInt();
        int sum=0;
        for (int i=1;i<=n;i++) {
            sum+=i;
        }
        System.out.println("The sum is "+sum);
    }
}
```

And now for the COBOL version:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SumOfIntegers.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 n   BINARY-LONG.
01 i   BINARY-LONG.
01 sum BINARY-LONG VALUE 0.
PROCEDURE DIVISION.
DISPLAY "Enter a positive integer"
ACCEPT n
PERFORM VARYING i FROM 1 BY 1 UNTIL i > n
    ADD i TO sum
END-PERFORM
DISPLAY "The sum is " sum.
```

My familiarity with COBOL may be prejudicing my opinion, but it doesn't appear to me that the Java code is any simpler than the COBOL code. In case you're interested in character counts, the Java code comes in at 278 (not counting indentation characters). The COBOL code is 298 (274 without the `IDENTIFICATION DIVISION.` header).

Despite what you've seen here, the more complex the programming logic being implemented, the more concise the Java code will appear to be, even compared to 2002-standard COBOL. That conciseness comes with a price though — program code readability. Java (or C or C++ or C#) programs are generally intelligible only to trained programmers. COBOL programs can, however, be quite understandable by non-programmers. This is actually a

side-effect of the "wordiness" of the language, where COBOL statements use natural English words to describe their actions. This inherent readability has come in handy many times throughout my career when I've had to learn obscure business (or legal) processes by reading the COBOL program code that supports them.

The "modern" languages, like Java, also have their own "boilerplate" infrastructure overhead that must be coded in order to write the logic that is necessary in the program. Take for example the `public static void main(String[] arg)` and `import java.util.Scanner;` statements. The critics tend to forget about this when they criticize COBOL for its structural "overhead".

When it first was developed, COBOL's easily-readable syntax made it profoundly different from anything that had been seen before. For the first time, it was possible to specify logic in a manner that was — at least to some extent — comprehensible even to non-programmers. Take for example, the following code written in FORTRAN — a language developed only a year before COBOL:

```
EXT = PRICE * IQTY
INVTOT = INVTOT + EXT
```

With its original limitation on the length of variable names (one- to six-character names comprised of a letter followed by up to five letters and/or digits), its implicit rule that variables were automatically created as real (floating-point) unless their name started with a letter in the range I-N, and its use of algebraic notation to express actions being taken, FORTRAN wasn't a particularly readable language, even for programmers. Compare this with the equivalent COBOL code:

```
MULTIPLY price BY quantity GIVING extended-amount
ADD extended-amount TO invoice-total
```

Clearly, even a non-programmer could at least conceptually understand what was going on! Over time, languages like FORTRAN evolved more robust variable names, and COBOL introduced a more formula-based syntactical capability for arithmetic operations, but FORTRAN was never as readable as COBOL.

Because of its inherent readability, I would MUCH rather be handed an assignment to make significant changes to a COBOL program about which I know nothing than to be asked to do the same with a C, C++, C# or Java program.

Those that argue that it is too boring / wasteful / time-consuming / insulting (pick one) to have to code a COBOL program "from scratch" are clearly ignorant of the following facts:

- Many systems have program-development tools available to ease the task of coding programs; those tools that concentrate on COBOL are capable of providing templates for much of the "overhead" verbiage of any program. . .
- Good programmers have — for decades — maintained their own skeleton "template" programs for a variety of program types; simply load a template into a text editor and you've got a good start to the program. . .
- Legend has it that there's actually only been ONE program ever written in COBOL, and all programs ever "written" thereafter were simply derivatives of that one. Although this is clearly intended as a (probably) bad joke, it is nevertheless close to the very

simple truth that many programmers "reuse" existing COBOL programs when creating new ones. There's certainly nothing preventing this from happening with programs written in other languages, but it does seem to happen more in COBOL shops. It's ironic that "code re-usability" is one of the arguments used to justify the existence of the "modern" languages.

2.1.5. Divisions Organize Programs

COBOL programs are structured into four major areas of coding, each with its own purpose. These four areas are known as divisions.

Each division may consist of a variety of sections and each section consists of one or more paragraphs. A paragraph consists of sentences, each of which consists of one or more statements.

This hierarchical structure of program components standardises the composition of all COBOL programs. Much of this manual describes the various divisions, sections, paragraphs and statements that may comprise any COBOL program.

2.1.6. Copybooks

`COPY` statement (see [COPY], page 63) A *Copybook* is a segment of program code that may be utilized by multiple programs simply by having those programs use the `COPY` statement to import that code. This code may define files, data structures or procedural code.

Today's current programming languages have a statement (usually, this statement is named "import", "include" or "#include") that performs this same function. What makes the COBOL copybook feature different than the "include" facility in newer languages, however, is the fact that the `COPY` statement can edit the imported source code as it is being copied. This capability makes copybook libraries extremely valuable to making code reusable. Also see section 3. Compiler Directing Facility commands COPY and REPLACE.

2.1.7. Structured Data

A contiguous area of storage within the memory space of a program that may be referenced, by name, in a COBOL program is referred to as a *Data Item*. Other programming languages use the term variable, property or attribute to describe the same thing.

COBOL introduced the concept of structured data. The principle of structured data in COBOL is based on the idea of being able to group related and contiguously-allocated data items together into a single aggregate data item, called a *Group Item*. For example, a 35-character 'Employee-Name' group item might consist of a 20-character 'Last-Name' followed by a 14-character 'First-Name' and a 1-character 'Middle-Initial'.

A data item that isn't itself formed from other data items is referred to in COBOL as an *Elementary Item*. In the previous example, 'Last-Name', 'First-Name' and 'Middle-Initial' are all elementary items.

2.1.8. Files

One of COBOL's strengths is the wide variety of data files it is capable of accessing. GnuCOBOL programs, like those created with other COBOL implementations, need to have the

structure of any files they will be reading and/or writing described to them. The highest-level characteristic of a file's structure is defined by specifying the organization of the file, as follows:

`ORGANIZATION LINE SEQUENTIAL`

These are files with the simplest of all internal structures. Their contents are structured simply as a series of identically- or differently-sized data records, each terminated by a special end-of-record delimiter character. An ASCII line-feed character (hexadecimal 0A) is the end-of-record delimiter character used by any UNIX or pseudo-UNIX (MinGW, Cygwin, OSX) GnuCOBOL build. A truly native Windows build would use a carriage-return, line-feed (hexadecimal 0D0A) sequence.

Records must be read from or written to these files in a purely sequential manner. The only way to read (or write) record number 100 would be to have read (or written) records number 1 through 99 first.

When the file is written to by a GnuCOBOL program, the delimiter sequence will be automatically appended to each data record as it is written to the file. A `WRITE` (see [WRITE], page 417) to this type of file will be done as if a `BEFORE ADVANCING 1 LINE` clause were specified on the `WRITE`, if no `ADVANCING` clause is coded.

When the file is read, the GnuCOBOL runtime system will strip the trailing delimiter sequence from each record. The data will be padded (on the right) with spaces if the data just read is shorter than the area described for data records in the program. If the data is too long, it will be truncated and the excess will be lost.

These files should not be defined to contain any exact binary data fields because the contents of those fields could inadvertently have the end-of-record sequence as part of their values — this would confuse the runtime system when reading the file, and it would interpret that value as an actual end-of-record sequence.

The following environment variables can have an effect on LINE SEQUENTIAL processing behaviour so may require changes to settings for:

```
COB_LS_FIXED
COB_LS_NULLS
COB_LS_SPLIT

When using Micro Focus compatable formats:
COB-MF-LS-NULLS
COB_MF_LS_INSTAB
COB_MF_LS_SPLIT
COB_MF_LS_VALIDATE
```

See sections 10.2.3.4 and 10.2.3.7 for more information.

`LINE ADVANCING`

These are files with an internal structure similar to that of a line sequential file. These files are defined (without an explicit `ORGANIZATION` specification)

using the `LINE ADVANCING` clause on their `SELECT` statement (see [SELECT], page 104).

When this kind of file is written to by a GnuCOBOL program, an end-of-record delimiter sequence will be automatically added to each data record as it is written to the file. A `WRITE` to this type of file will be done as if an `AFTER ADVANCING 1 LINE` clause were specified on the `WRITE`, if no `ADVANCING` clause is coded.

Like line sequential files, these files should not be defined to contain any exact binary data fields because the contents of those fields could inadvertently have the end-of-record sequence as part of their values — this would confuse the runtime system when reading the file, and it would interpret that value as an actual end-of-record sequence.

`ORGANIZATION SEQUENTIAL`

These files also have a simple internal structure. Their contents are structured simply as an arbitrarily-long sequence of data characters. This sequence of characters will be treated as a series of fixed-length records simply by logically splitting the sequence of characters up into fixed-length segments, each as long as the maximum record size defined in the program. There are no special end-of-record delimiter characters in the file and when the file is written to by a GnuCOBOL program, no delimiter sequence is appended to the data.

Records in this type of file are all the same physical length, except possibly for the very last record in the file, which may be shorter than the others. If variable-length logical records are defined to the program, the space occupied by each physical record in the file will occupy the space described by the longest record description in the program.

So, if a file contains 1275 characters of data, and a program defines the structure of that file as containing 100-character records, then the file contents will consist of twelve (12) 100-character records with a final record containing only 75 characters.

It would appear that it should be possible to locate and process any record in the file directly simply by calculating its starting character position based upon the program-defined record size. Even so, however, records must be still be read or written to these files in a purely sequential manner. The only way to read (or write) record number 100 would be to have read (or written) records number 1 through 99 first.

When the file is read, the data is transferred into the program exactly as it exists in the file. In the event that a short record is read as the very last record, that record will be padded (to the right) with spaces.

Care must be taken that programs reading such a file describe records whose length is exactly the same as that used by the program that created the file. For example, the following shows the contents of a `SEQUENTIAL` file created by a program that wrote five 6-character records to it. The '`A`', '`B`', ... values reflect the records that were written to the file:

'`AAAAAA`'

'BBBBBB'
'CCCCCC'
'DDDDDD'
'EEEEEE'

Now, assume that another program reads this file, but describes 10-character records rather than 6. Here are the records *that* program will read:

'AAAAAABBBB'
'BBCCCCCCDD'
'DDDDEEEEEE'

There may be times where this is exactly what you were looking for. More often than not, however, this is not desirable behaviour. Suggestion: use a copybook to describe the record layouts of any file; this guarantees that multiple programs accessing that file will "see" the same record sizes and layouts by coding a `COPY` statement (see [COPY], page 63) to import the record layout(s) rather than hand-coding them.

These files *can* contain exact binary data fields. Because there is no character sequence that constitutes an end-of-record delimiter, the contents of record fields are irrelevant to the reading process.

`ORGANIZATION RELATIVE`

The contents of these files consist of a series of fixed-length data records prefixed with a four-byte record header. The record header contains the length of the data, in bytes. The byte-count does not include the four-byte record header.

Records in this type of file are all the same physical length. If variable-length logical records are defined to the program, the space occupied by each physical record in the file will occupy the maximum possible space, and the logical record length field will contain the number of bytes of data in the record that are actually in use.

This file organization was defined to accommodate either sequential or random processing. With a `RELATIVE` file, it is possible to read or write record 100 directly, without having to have first read or written records 1-99. The GnuCOBOL runtime system uses the program-defined maximum record size to calculate a relative byte position in the file where the record header and data begin, and then transfers the necessary data to or from the program.

When the file is written by a GnuCOBOL program, no delimiter sequence is appended to the data, but a record-length field is added to the beginning of each physical record.

When the file is read, the data is transferred into the program exactly as it exists in the file.

Care must be taken that programs reading such a file describe records whose length is exactly the same as that used by the programs that created the file. It won't end well if the GnuCOBOL runtime library interprets a four-byte ASCII character string as a record length when it transfers data from the file into the program!

Suggestion: use a copybook to describe the record layouts of any file; this guarantees that multiple programs accessing that file will "see" the same record sizes and layouts by coding a `COPY` statement (see [COPY], page 63) to import the record layout(s) rather than hand-coding them.

These files can contain exact binary data fields. The contents of record fields are irrelevant to the reading process as there is no end-of-record delimiter.

`ORGANIZATION INDEXED`

This is the most advanced file structure available to GnuCOBOL programs. It's not possible to describe the physical structure of such files because that structure will vary depending upon which advanced file-management facility was included into the GnuCOBOL build you will be using (Berkeley Database [BDB], VBISAM, etc.). We will — instead — discuss the logical structure of the file.

There will be multiple structures stored for an `INDEXED` file. The first will be a data component, which may be thought of as being similar to the internal structure of a relative file. Data records may not, however, be directly accessed by their record number as would be the case with a relative file, nor may they be processed sequentially by their physical sequence in the file.

The remaining structures will be one or more index components. An index component is a data structure that (somehow) enables the contents of a field, called a primary key, within each data record (a customer number, an employee number, a product code, a name, etc.) to be converted to a record number so that the data record for any given primary key value can be directly read, written and/or deleted. Additionally, the index data structure is defined in such a manner as to allow the file to be processed sequentially, record-by-record, in ascending sequence of the primary key field values. Whether this index structure exists as a binary-searchable tree structure (b-tree), an elaborate hash structure or something else is pretty much irrelevant to the programmer — the behaviour of the structure will be as it was just described. The actual mechanism used will depend upon the advanced file-management package was included into your GnuCOBOL implementation when it was built.

The runtime system will not allow two records to be written to an indexed file with the same primary key value.

The capability exists for an additional field to be defined as what is known as an alternate key. Alternate key fields behave just like primary keys, allowing both direct and sequential access to record data based upon the alternate key field values, with one exception. That exception is the fact that alternate keys may be allowed to have duplicate values, depending upon how the alternate key field is described to the GnuCOBOL compiler.

There may be any number of alternate keys, but each key field comes with a disk space penalty as well as an execution time penalty. As the number of alternate key fields increases, it will take longer and longer to write and/or modify records in the file.

These files can contain exact binary data fields. The contents of record fields are irrelevant to the reading process as there is no end-of-record delimiter.

All files are initially described to a GnuCOBOL program using a `SELECT` statement (see [SELECT], page 104). In addition to defining a name by which the file will be referenced within the program, the `SELECT` statement will specify the name and path by which the file will be known to the operating system along with its organization, locking and sharing attributes.

A file description in the `FILE SECTION` (see [FILE SECTION], page 123) will define the structure of records within the file, including whether or not variable-length records are possible and, if so, what the minimum and maximum length might be. In addition, the file description entry can specify file I/O block sizes.

2.1.9. Table Handling

Other programming languages have arrays; COBOL has tables. They're basically the same thing. There are two special statements that exist in the COBOL language — `SEARCH` and `SEARCH ALL` — that make finding data in a table easy.

`SEARCH` searches a table sequentially, stopping only when either a table entry matching one of any number of search conditions is found, or when all table entries have been checked against the search criteria and none matched any of those criteria.

`SEARCH ALL` performs an extremely fast search against a table sorted by a key field contained in each table entry. The algorithm used for such a search is a binary search. The algorithm ensures that only a small number of entries in the table need to be checked in order to find a desired entry or to determine that the desired entry doesn't exist in the table. The larger the table, the more effective this search becomes. For example, a binary search of a table containing 32,768 entries will locate a particular entry or determine the entry doesn't exist by looking at no more than fifteen (15) entries! The algorithm is explained in detail in the documentation of the `SEARCH ALL` statement (see [SEARCH ALL], page 379).

Finally, COBOL has the ability to perform in-place sorts of the data that is found in a table.

2.1.10. Sorting and Merging Data

The COBOL language includes a powerful `SORT` statement that can sort large amounts of data according to arbitrarily complex key structures. This data may originate from within the program or may be contained in one or more external files. The sorted data may be written automatically to one or more output files or may be processed, record-by-record in the sorted sequence.

A companion statement — `MERGE` — can combine the contents of multiple files together, provided those files are all pre-sorted in a similar manner according to the same key structure. The resulting output will consist of the contents of all of the input files, merged together and sequenced according to the common key structure(s). The output generated by a `MERGE` statement may be written automatically to one or more output files or may be processed internally by the program.

A special form of the SORT statement also exists just to sort the data that resides in a table. This is particularly useful if you wish to use SEARCH ALL against the table.

2.1.11. String Manipulation

There have been programming languages designed specifically for the processing of text strings, and there have been programming languages designed for the sole purpose of performing high-powered numerical computations. Most programming languages fall somewhere in the middle.

COBOL is no exception, although it does include some very powerful string manipulation capabilities; GnuCOBOL actually has even more string-manipulation capabilities than many other COBOL implementations. The following summarizes GnuCOBOL's string-processing capabilities:

Concatenate two or more strings

- CONCATENATE intrinsic function (see [CONCATENATE], page 437).
- STRING statement (see [STRING], page 401).

Conversion of a numeric time or date to a formatted character string

- LOCALE-TIME intrinsic function (see [LOCALE-TIME], page 473).
- LOCALE-DATE intrinsic function (see [LOCALE-DATE], page 472).

Convert a binary value to its corresponding character in the program's character set

- CHAR intrinsic function (see [CHAR], page 434). Add 1 to argument before invoking the function; the description of the CHAR intrinsic function presents a technique utilizing the MOVE statement that will accomplish the same thing without the need of adding 1 to the numeric argument value first.

Convert a character string to lower-case

- LOWER-CASE intrinsic function (see [LOWER-CASE], page 477).
- C$TOLOWER built-in system subroutine (see [C$TOLOWER], page 556).
- CBL_TOLOWER built-in system subroutine (see [CBL_TOLOWER], page 596).

Convert a character string to upper-case

- UPPER-CASE intrinsic function (see [UPPER-CASE], page 529).
- C$TOUPPER built-in system subroutine (see [C$TOUPPER], page 557).
- CBL_TOUPPER built-in system subroutine (see [CBL_TOUPPER], page 596).

Convert a character string to only printable characters

- C$PRINTABLE built-in system subroutine (see [C$PRINTABLE], page 554).

Convert a character to its numeric value in the program's character set

- ORD intrinsic function (see [ORD], page 501). Subtract 1 from the result; the description of the ORD intrinsic function presents a technique utilizing the MOVE statement that will accomplish the same thing without the need of adding 1 to the numeric argument value first.

Count occurrences of sub strings in a larger string

- `INSPECT` statement (see [INSPECT], page 344) with the `TALLYING` clause.

Decode a formatted numeric string back to a numeric value

- `NUMVAL` intrinsic function (see [NUMVAL], page 496).
- `NUMVAL-C` intrinsic function (see [NUMVAL-C], page 497).

Determine the length of a string or data-item capable of storing strings

- `LENGTH` intrinsic function (see [LENGTH], page 469).
- `BYTE-LENGTH` intrinsic function (see [BYTE-LENGTH], page 433).

Extract a sub string from a string based on its starting character position and length

- Use of a reference modifier on the string field - See [Reference Modifiers], page 40.

Format a numeric item for output, including thousands-separators ('`,`' in the USA), currency symbols ('`$`' in the USA), decimal points, credit/Debit Symbols, Leading Or Trailing Sign Characters

- `MOVE` statement (see [MOVE], page 352) with picture-symbol editing applied to the receiving field:

Justification (left, right or centred) of a string field

- `C$JUSTIFY` built-in system subroutine (see [C$JUSTIFY], page 550).

Monoalphabetic substitution of one or more characters in a string with different characters

- `INSPECT` statement (see [INSPECT], page 344) with the `CONVERTING`.
- `TRANSFORM` statement (see [TRANSFORM], page 411).
- `SUBSTITUTE` intrinsic function (see [SUBSTITUTE], page 518).
- `SUBSTITUTE-CASE` intrinsic function (see [SUBSTITUTE-CASE], page 519).

Parse a string, breaking it up into sub strings based upon one or more delimiting character sequences[1]

- `UNSTRING` statement (see [UNSTRING], page 413).

Removal of leading or trailing spaces from a string

- `TRIM` intrinsic function (see [TRIM], page 528).

Substitution of a single sub string with another of the same length, based upon the sub strings starting character position and length

- `MOVE` statement (see [MOVE], page 352) with a reference modifier on the "receiving" field (see [Reference Modifiers], page 40).

Substitution of one or more sub strings in a string with replacement sub strings of the same length, regardless of where they occur

- `INSPECT` statement (see [INSPECT], page 344) with a `REPLACING` clause.
- `SUBSTITUTE` intrinsic function (see [SUBSTITUTE], page 518).

[1] These delimiters may be single characters, multiple-character strings or multiple consecutive occurrences of either

- `SUBSTITUTE-CASE` intrinsic function (see [SUBSTITUTE-CASE], page 519).

Substitution of one or more sub strings in a string with replacement sub strings of a potentially different length, regardless of where they occur

- `SUBSTITUTE` intrinsic function (see [SUBSTITUTE], page 518).

- `SUBSTITUTE-CASE` intrinsic function (see [SUBSTITUTE-CASE], page 519).

2.1.12. Screen Formatting Features

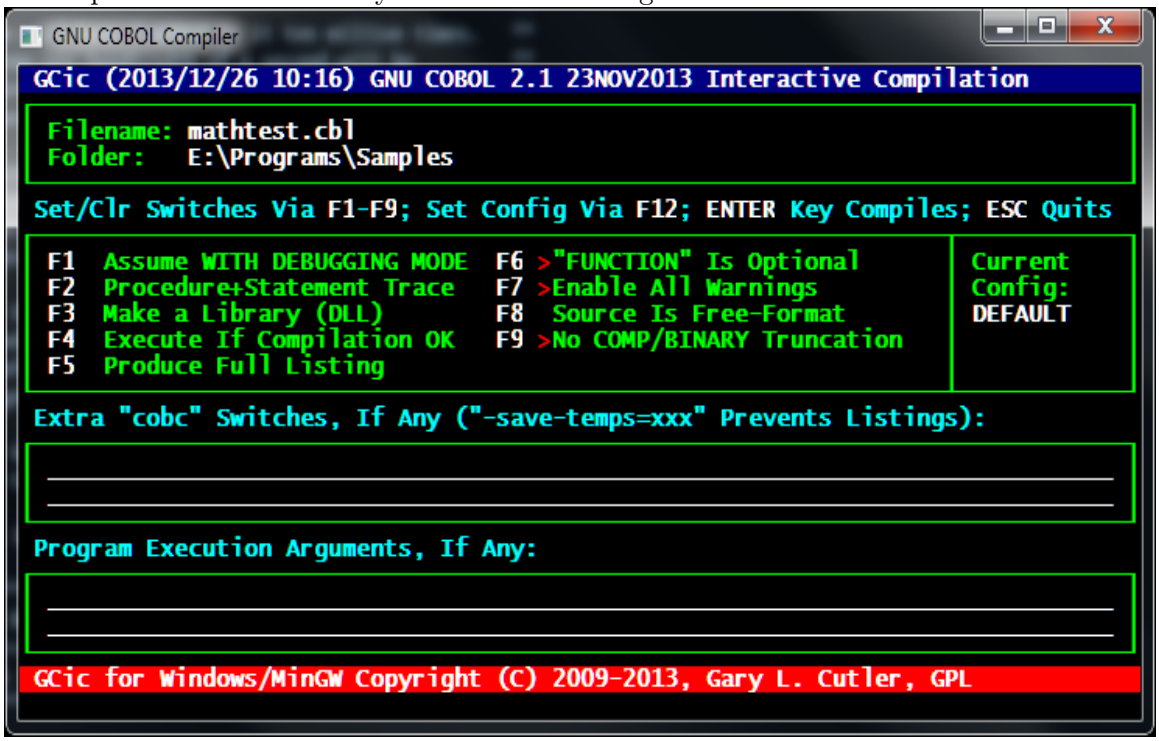
The COBOL2002 standard formalizes extensions to the COBOL language that allow for the definition and processing of text-based screens, as is a typical function on mainframe and midframe computers as well as on many point-of-sale (i.e. "cash register") systems. GnuCOBOL implements virtually all the screen-handling features described by COBOL2002.

These features allow fields to be displayed at specific row/column positions, various colors and video attributes to be assigned to screen fields and the pressing of specific function keys (F1, F2, ...) to be detectable. All of this takes place through the auspices of the `SCREEN SECTION` (see [SCREEN SECTION], page 143) and special formats of the `ACCEPT` statement (see [ACCEPT], page 268) and the `DISPLAY` statement (see [DISPLAY], page 305).

The COBOL2002 standard, and therefore GnuCOBOL, only covers textual user interface (TUI) screens (those comprised of ASCII characters presented using a variety of visual attributes) and not the more-advanced graphical user interface (GUI) screen design and processing capabilities built into most modern operating systems. There are subroutine-based packages available that can do full GUI presentation — most of which may be called by GnuCOBOL programs, with a moderate research time investment (Tcl/Tk, for example) — but none are currently included with GnuCOBOL.

2.1.12.1. A Sample Screen

A Sample Screen Produced by a GnuCOBOL Program:



The above screen was produced by the GnuCOBOL Interactive Compiler, or GCic. See Section "GCic" in *GnuCOBOL Sample Programs*, for the source and cross-reference listing of this program.

Screens are defined in the screen section of the data division. Once defined, screens are used at run-time via the `ACCEPT` and `DISPLAY` statements.

2.1.12.2. Color Palette and Video Attributes

GnuCOBOL supports the following visual attribute specifications in the `SCREEN SECTION` (see [SCREEN SECTION], page 143):

Color
: Eight (8) different colors may be specified for both the background (screen) and foreground (text) color of any row/column position on the screen. Colors are specified by number, although a copybook supplied with all GnuCOBOL distributions (`screenio.cpy`) defines `COB-COLOR-xxxxxx` names for the various colors so they may be specified as a more meaningful name rather than a number. The eight colors, by number, with the constant names defined in `screenio.cpy`, are as follows:

`Black`	COB-COLOR-BLACK
`Blue`	COB-COLOR-BLUE
`Green`	COB-COLOR-GREEN

`Cyan`	COB-COLOR-CYAN
`Red`	COB-COLOR-RED
`Magenta`	COB-COLOR-MAGENTA
`Yellow`	COB-COLOR-YELLOW
`White`	COB-COLOR-WHITE

Text Brightness
: There are three possible brightness levels supported for text — lowlight (dim), normal and highlight (bright). Not all GnuCOBOL implementations will support all three (some treat lowlight the same as normal). The deciding factor as to whether two or three levels are supported lies with the version of the *curses* package that is being used. This is a utility screen-IO package that is included into the GnuCOBOL run-time library when the GnuCOBOL software is built.

  As a general rule of thumb, Windows implementations support two levels while Unix ones support all three.

Blinking
: This too is a video feature that is dependent upon the *curses* package built into your version of GnuCOBOL. If blinking is enabled in that package, text displayed in fields defined in the screen section as being blinking will endlessly cycle between the brightest possible setting (highlight) and an "invisible" setting where the text color matches that of the field background color. A Windows build, which generally uses the "pcurses" package, will uses a brighter-than-normal background color to signify "blinking".

Reverse Video
: This video attribute simply swaps the foreground and background colors and display options.

Field Outlining
: It is possible, if supported by the *curses* package being used, to draw borders on the top, left and/or bottom edges of a field.

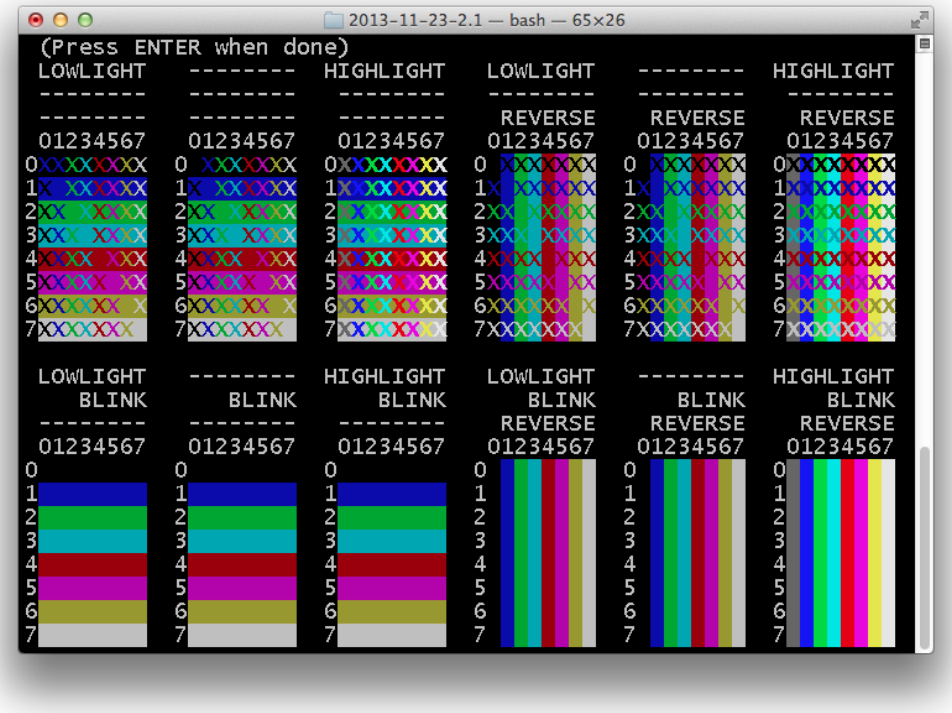
Secure Input
: If desired, screen fields used as input fields may defined as "secure" fields, where each input character (regardless of what was actually typed) will appear as an asterisk (*) character. The actual character whose key was pressed will still be stored into the field in the program, however. This is very useful for password or account number fields.

Prompt Character
: Input fields may have any character used as a fill character. These fill characters provide a visual indication of the size of the input field, and will automatically be transformed into spaces when the input field is processed by the program. If no such character is defined for an input field, an underscore ('_') will be assumed.

The following is a sample of the GnuCOBOL color Palette, showing all possible combinations of the various video attributes. This example was prepared on a Macintosh running

OSX Mavericks (10.9). Blinking works — the screen snapshot shows things in mid blink, when the text and background colors are momentarily the same. Unfortunately, only two screen intensities are available (like Windows, the "lowlight" setting is the same as the default).

The GnuCOBOL Color Palette and Video Options::



The rows of each block are numbered with the background color while columns are numbered with the foreground color.

See Section "Colors" in *GnuCOBOL Sample Programs*, for a source and cross-reference listing of the program (Colors.cbl) that produced the above screen.

2.1.13. Report Writer Features

GnuCOBOL includes an implementation of the Report Writer Control System, or RWCS. The reportwriter module is now fully implemented as of version 3.0. This is a standardized, optional add-on feature to the COBOL language which automates much of the mechanics involved in the generation of printed reports by:

1. Controlling the pagination of reports, including:
   A. The automatic production of a one-time notice on the first page of the report (report heading).
   B. The production of zero or more header lines at the top of every page of the report (page heading).

C. The production of zero or more footer lines at the bottom of every page of the report (page footing).
D. The automatic numbering of printed pages.
E. The formatting of those report lines that make up the main body of the report (detail).
F. Full awareness of where the "pen" is about to "write" on the current page, automatically forcing an eject to a new page, along with the automatic generation of a page footer to close the old page and/or a page header to begin the new one.
G. The production of a one-time notice at the end of the last page of a report (report footing).

2. Performing special reporting actions based upon the fact that the data being used to generate the report has been sorted according to one or more key fields:
   A. Automatically suppressing the presentation of one or more fields of data from the detail group when the value(s) of the field(s) duplicate those of the previously generated detail group. Fields such as these are referred to as group-indicate fields.
   B. Automatically causing suppressed detail group-indicate fields to re-appear should a detail group be printed on a new page.
   C. Recognizing when control fields on the report — fields tied to those that were used as `SORT` statement (see [SORT], page 391) keys — have changed. This is known as a control break. The RWCS can automatically perform the following reporting actions when a control break occurs:
      - Producing a footer, known as a control footing after the detail lines that shared the same old value for the control field.
      - Producing a header, known as a control heading before the detail lines that share the same new value for the control field.
3. Perform data summarise, as follows:
   A. Automatically generating subtotals in control and/or report footings, summarizing values of any fields in the detail group.
   B. Automatically generating crossfoot totals in detail groups. These would be sums of two or more values presented in the detail group.

The `REPORT SECTION` (see [REPORT SECTION], page 135) documentation explores the description of reports and the `PROCEDURE DIVISION` (see [PROCEDURE DIVISION], page 247) chapter documents the various language statements that actually produce reports. Before reading these, you might find it helpful to read [Report Writer Usage], page 605, which is dedicated to putting the pieces together for you.

2.1.14. Data Initialization

There are three ways in which data division data gets initialized.

1. When a program or subprogram is first executed, much of the data in its data division will be initialized as follows:
   - Alphanumeric and alphabetic (i.e. text) data items will be initialized to `SPACES`.
   - Numeric data items will be initialized to a value of `ZERO`.

- Data items with an explicit `VALUE` (see [VALUE], page 244) clause in their definition will be initialized to that specific value.

The various sections of the data division each have their own rules as to when the actions described above will occur — consult the documentation on those sections for additional information.

These default initialization rules can vary quite substantially from one COBOL implementation to another. For example, it is quite common for data division storage to be initialized to all binary zeros except for those data items where `VALUE` clauses are present. Take care when working with applications originally developed for another COBOL implementation to ensure that GnuCOBOL's default initialization rules won't prove disruptive.

2. A programmer may use the `INITIALIZE` statement (see [INITIALIZE], page 339) to initialise any group or elementary data item at any time. This statement provides far more initialization options than just the simple rules stated above.
3. When the `ALLOCATE` statement (see [ALLOCATE], page 290) statement is used to allocate a data item or to simply allocate an area of storage of a size specified on the `ALLOCATE`, that allocation may occur with or without initialization, as per the programmer's needs.

2.1.15. Syntax Diagram Conventions

Syntax of the GnuCOBOL language will be described in special *syntax diagrams* using the following syntactical-description techniques:

`MANDATORY-RESERVED-WORD`
`~~~~~~~~~~~~~~~~~~~~~~~`

Reserved words of the COBOL language will appear in UPPER-CASE. When they appear underlined, as this one is, they are required reserved words.

`OPTIONAL-RESERVED-WORD`

When reserved words appear without underlining, as this one is, they are optional; such reserved words are available in the language syntax merely to improve readability — their presence or absence has no effect upon the program.

`ABBREVIATION`
`~~~~`

When only a portion of a reserved word is underlined, it indicates that the word may either be coded in its full form or may be abbreviated to the portion that is underlined.

`substitutable-items`

Generic terms representing user-defined substitutable items will be shown entirely in lower-case in syntax diagrams. When such items are referenced in text, they will appear as *substitutable-items*.

`Complex-Syntax-Clause`

Items appearing in Mixed Case within a syntax diagram represent complex clauses of other syntax elements that may appear in that position. Some COBOL syntax gets quite complicated, and using a convention such as this

significantly reduces the complexity of a syntax diagram. When such items are referenced in text, they will appear as *Complex-Syntax-Clause*.

`[ ]` Square bracket meta characters on syntax diagrams document language syntax that is optional. The [] characters themselves should not be coded. If a syntax diagram contains '`a [b] c`', the '`a`' and '`c`' syntax elements are mandatory but the '`b`' element is optional.

`|` Vertical bar meta characters on syntax diagrams document simple choices. The `|` character itself should not be coded. If a syntax diagram contains '`a|b|c`', exactly one of the items '`a`', '`b`' or '`c`' must be selected.

```
{ xxxxxx }
{ yyyyyy }
{ zzzzzz }
```

A vertical list of items, bounded by multiple brace characters, is another way of signifying a choice between a series of items where exactly one item must be selected. This form is used to show choices when one or more of the selections is more complex than just a single word, or when there are too many choices to present horizontally with '`|`' meta characters.

```
| xxxxxx |
| yyyyyy |
| zzzzzz |
```

A vertical list of items, bounded by multiple vertical bar characters, signifies a choice between a series of items where one *or more* of the choices could be selected.

`...` The ... meta character sequence signifies that the syntax element immediately preceding it may be repeated. The ... sequence itself should not be coded. If a syntax diagram contains `a b... c`, syntax element '`a`' must be followed by at least one '`b`' element (possibly more) and the entire sequence must be terminated by a '`c`' syntax element.

`{ }` The braces ('`{`' and '`}`') meta characters may be used to group a sequence of syntax elements together so that they may be treated as a single entity. The `{}` characters themselves should not be coded. These are typically used in combination with the '`|`' or '`...`' meta characters.

`$*^()-+=:"'<,>./`

Any of these characters appearing within a syntax diagram are to be interpreted literally, and are characters that must be coded — where allowed — in the statement whose format is being described. Note that a '`.`' character is a literal character that must be coded on a statement whereas a '`...`' symbol is the meta character sequence described above.

2.1.16. Format of Program Source Lines

Prior to the COBOL2002 standard, source statements in COBOL programs were structured around 80-column punched cards. This means that each source line in a COBOL program consisted of five different "areas", defined by their column number(s).

As of the COBOL2002 standard, a second mode now exists for COBOL source code statements — in this mode of operation, COBOL statements may each be up to 255 characters long, with no specific requirements as to what should appear in which columns.

Of course, in keeping with the long-standing COBOL tradition of maintaining backwards compatibility with older standards, programmers (and, of course, compliant COBOL compilers) are capable of working in either mode. It is even possible to switch back and forth in the same program. The terms *Fixed Format Mode* and *Free Format Mode* are used to refer to these two modes of source code formatting.

The GnuCOBOL compiler (cobc) supports both of these source line format modes, defaulting to Fixed Format Mode lacking any other information.

The compiler can be instructed to operate in either mode in any of the following four ways:

1. Using a compiler option switch — use the `-fixed` switch to start in Fixed Format Mode (remember that this is the default) or the `-free` switch to start in Free Format Mode.
2. You may use the `SOURCEFORMAT AS FIXED` and `SOURCEFORMAT AS FREE` clauses of the `>>SET` CDF directive (see [>>SET], page 71) within your source code to switch to Fixed or Free Format Mode, respectively.
3. You may use the `>>FORMAT IS FIXED` and `FORMAT IS FREE` clauses of the `>>DEFINE` CDF directive (see [>>DEFINE], page 68) within your source code to switch to Fixed or Free Format Mode, respectively.
4. You may use the `>>SOURCE` CDF directive (see [>>SOURCE], page 73) to switch to Free Format Mode (`>>SOURCE FORMAT IS FREE`) or Fixed Format Mode (`>>SOURCE FORMAT IS FIXED`.

Using methods 2-4 above, you may switch back and forth between the two formats at will.

The last three options above are all equivalent; all three are supported by GnuCOBOL so that source code compatibility may be maintained with a wide variety of other COBOL implementations. With all three, if the compiler is *currently* in Fixed Format Mode, the `>>` must begin in column 8 or beyond, provided no part of the directive extends past column 72. If the compiler is currently in Free Format Mode, the `>>` may appear in any column, provided no part of the directive extends past column 255.

Depending upon which source format mode the compiler is in, you will need to follow various rules for the format mode currently in effect. These rules are presented in the upcoming paragraphs.

The following discussion presents the various components of every GnuCOBOL source line record when the compiler is operating in Fixed Format Mode. Remember that this is the default mode for the GnuCOBOL compiler.

1-6 Sequence Number Area

Historically, back in the days when punched-cards were used to submit COBOL program source to a COBOL compiler, this part of a COBOL statement was reserved for a six-digit sequence number. While the contents of this area are ignored by COBOL compilers, it existed so that a program actually punched

on 80-character cards could — if the card deck were dropped on the floor — be run through a card sorter machine and restored to its proper sequence. Of course, this isn't necessary today; if truth be told, it hasn't been necessary for a long time.

See [Marking Changes in Programs], page 699, for discussion of a valuable use to which the sequence number area may be put today.

7 Indicator Area

Column 7 serves as an indicator in which one of five possible values will appear — space, `D` (or `d`), `-` (dash), `/` or `*`. The meanings of these characters are as follows:

space — No special meaning — this is the normal character that will appear in this area.

D/d — The line contains a valid GnuCOBOL statement that is normally treated as a comment unless the program is being compiled in debugging mode.

* — The line is a comment.

/ — The line is a comment that will also force a page eject in the compilation listing. While GnuCOBOL will honour such a line as a comment, it will not form-feed any generated listing.

\- — The line is a continuation of the previous line. These are needed only when an alphanumeric literal (quoted character string), reserved word or user-defined word are being split across lines.

8-11 Area A

Language `DIVISION`, `SECTION` and paragraph section headers must begin in Area A, as must the level numbers 01, 77 in data description entries and the `FD` and `SD` file and `SORT` description headers.

12-72 Area B

All other COBOL programming language components are coded in these columns.

73-80 Program Name Area

This is another obsolete area of COBOL statements. This part of every statement also hails back to the day when programs were punched on cards; it was expected that the name of the program (or at least the first 8 characters of it) would be punched here so that — if a dropped COBOL source deck contained more than one program — that handy card sorter machine could be used to first separate the cards by program name and then sort them by sequence number. Today's COBOL compilers (including GnuCOBOL) simply ignore anything past column 72.

See [Marking Changes in Programs], page 699, for discussion of a valuable use to which the program name area may be put today.

2.1.17. Program Structure

Complete GnuCOBOL Program Syntax

```
[ IDENTIFICATION DIVISION. ]
  ~~~~~~~~~~~~~~~~~~~~~~~
  PROGRAM-ID|FUNCTION-ID.  name-1 [ Program-Options ] .
  ~~~~~~~~~~ ~~~~~~~~~~~
[ ENVIRONMENT DIVISION. ]
  ~~~~~~~~~~~ ~~~~~~~~
[ CONFIGURATION SECTION. ]
  ~~~~~~~~~~~~~ ~~~~~~~
[ SOURCE-COMPUTER.         Compilation-Computer-Specification .  ]
  ~~~~~~~~~~~~~~~
[ OBJECT-COMPUTER.         Execution-Computer-Specification .  ]
  ~~~~~~~~~~~~~~~
[ REPOSITORY.              Function-Specification...  .  ]
  ~~~~~~~~~~
[ SPECIAL-NAMES.           Program-Configuration-Specification .  ]
  ~~~~~~~~~~~~~
[ INPUT-OUTPUT SECTION. ]
  ~~~~~~~~~~~~ ~~~~~~~
[ FILE-CONTROL.            General-File-Description...  .  ]
  ~~~~~~~~~~~~
[ I-O-CONTROL.             File-Buffering-Specification...  .  ]
  ~~~~~~~~~~~
[ DATA DIVISION. ]
  ~~~~~~~~~~~~~
[ FILE SECTION.            Detailed-File-Description...  .  ]
  ~~~~~~~~~~~~
[ WORKING-STORAGE SECTION. Permanent-Data-Definition...  .  ]
  ~~~~~~~~~~~~~~~ ~~~~~~~
[ LOCAL-STORAGE SECTION.   Temporary-Data-Definition...  .  ]
  ~~~~~~~~~~~~~ ~~~~~~~
[ LINKAGE SECTION.         Subprogram-Argument-Description...  .  ]
  ~~~~~~~ ~~~~~~~
[ REPORT SECTION.          Report-Description...  .  ]
  ~~~~~~ ~~~~~~~
[ SCREEN SECTION.          Screen-Layout-Definition...  .  ]
  ~~~~~~ ~~~~~~~
  PROCEDURE DIVISION [ { USING  Subprogram-Argument...      } ]
  ~~~~~~~~~ ~~~~~~~~   { ~~~~~                              }
                       { CHAINING Main-Program-Argument...  }
                         ~~~~~~~~
```

```
                     [   RETURNING identifier-1 ] .
[ DECLARATIVES. ]        ~~~~~~~~~
  ~~~~~~~~~~~~
[ Event-Handler-Routine...  .  ]
[ END DECLARATIVES. ]
  ~~~ ~~~~~~~~~~~~
  General-Program-Logic
[ Nested-Subprogram...  ]
[ END PROGRAM|FUNCTION name-1 ]
  ~~~ ~~~~~~~ ~~~~~~~~
```

Each program consists of up to four *Divisions* (major groupings of sections, paragraphs and descriptive or procedural coding that all relate to a common purpose), named Identification, Environment, Data and Procedure.

1. Not all divisions are needed in every program, but they must be specified in the order shown when they *are* used.
2. The following points pertain to the identification division
   - The `IDENTIFICATION DIVISION.` header is always optional.
3. The following points pertain to the environment division:
   - If both optional sections of this division are coded, they must be coded in the sequence shown.
   - Each of these sections consists of a series of specific paragraphs (`SOURCE-COMPUTER` and `OBJECT-COMPUTER`, for example). Each of these paragraphs serves a specific purpose. If no code is required for the purpose one of the paragraphs serves, the entire paragraph may be omitted.
   - If none of the paragraphs within one of the sections are coded, the section header itself may be omitted.
   - The paragraphs within each section may only be coded in that section, but may be coded in any order.
   - If none of the sections within the environment division are coded, the `ENVIRONMENT DIVISION.` header itself may be omitted.
4. The following points pertain to the data division:
   - The data division consists of six optional sections — when used, those sections must be coded in the order shown in the syntax diagram.
   - Each of these sections consists of code which serves a specific purpose. If no code is required for the purpose one of those sections serves, the entire section, including its header, may be omitted.
   - If none of the sections within the data division are coded (a highly unlikely, but theoretically possible circumstance), the `DATA DIVISION.` header itself may be omitted.
5. The following points pertain to the procedure division:

- As with the other divisions, the procedure division may consist of sections and those sections may — in turn — consist of paragraphs. Unlike the other divisions, however, section and paragraph names are defined by the programmer, and there may not be any defined at all if the programmer so wishes.

- Each Event-Handler-Routine will be a separate section devoted to trapping a particular run-time event. If there are no such sections coded, the `DECLARATIVES.` and `END DECLARATIVES.` lines may be omitted.

6. A single file of COBOL source code may contain:

   - A portion of a program; these files are known as copybooks

   - A single program. In this case, the `END PROGRAM` or `END FUNCTION` statement is optional.

   - Multiple programs, separated from one another by `END PROGRAM` or `END FUNCTION` statements. The final program in such a source code file need not have an `END PROGRAM` or `END FUNCTION` statement.

7. Subprogram '`B`' may be nested inside program '`A`' by including program B's source code at the end of program A's procedure division without an intervening `END PROGRAM A.` or `END FUNCTION A.` statement. For now, that's all that will be said about nesting. See [Independent vs Contained vs Nested Subprograms], page 679, for more information.

8. Regardless of how many programs comprise a single GnuCOBOL source file, only a single output executable program will be generated from that source file when the file is compiled.

2.1.18. Comments

The following information describes how comments may be embedded into GnuCOBOL program source to provide documentation.

Comment Type	**Source Mode — Description**
Blank Lines	**FIXED** — Blank lines may be inserted as desired. **FREE** — Blank lines may be inserted as desired.
Full-line comments	**FIXED** — An entire source line will be treated as a comment (and will be ignored by the compiler) by coding an asterisk ('`*`') in column seven (7). **FREE** — An entire source line will be treated as a comment (and will be ignored by the compiler) by coding the sequence '`*>`', starting in any column, as the first non-blank characters on the line.

Full-line comments with form-feed	**FIXED** — An entire source line will be treated as a comment by coding a slash ('`/`') in column seven (7). Many COBOL compilers will also issue a form-feed in the program listing so that the '`/`' line is at the top of a new page. The GnuCOBOL compiler does not support this form-feed behaviour. The GnuCOBOL Interactive Compiler, or GCic, *does* support this form-feed behaviour when it generates program source listings! See Section "GCic" in *GnuCOBOL Sample Programs*, for the source and cross-reference listing (produced by GCic) of this program — you can see the effect of '`/`' there. **FREE** — There is no Free Source Mode equivalent to '`/`'.
Partial-line comments	**FIXED** — Any text following the character sequence '`*>`' on a source line will be treated as a comment. The '`*`' must appear in column seven (7) or beyond. **FREE** — Any text following the character sequence '`*>`' on a source line will be treated as a comment. The '`*`' may appear in any column.
Comments that may be treated as code, typically for debugging purposes	**FIXED** — By coding a '`D`' in column 7 (upper- or lower-case), an otherwise valid GnuCOBOL source line will be treated as a comment by the compiler. **FREE** — By specifying the character sequence '`>>D`' (upper- or lower-case) as the first non-blank characters on a source line, an otherwise valid GnuCOBOL source line will be treated as a comment by the compiler. Debugging statements may be compiled either by specifying the `-fdebugging-line` switch on the GnuCOBOL compiler or by adding the `WITH DEBUGGING MODE` clause to the `SOURCE-COMPUTER` paragraph.

2.1.19. Literals

Literals are constant values that will not change during the execution of a program. There are two fundamental types of literals — numeric and alphanumeric.

2.1.19.1. Numeric Literals

A numeric literal

- Integers such as 1, 56, 2192 or -54.
- Non-integer fixed point values such as 1.317 or -2.95.
- Floating-point values using '`Enn`' notation such as `9.92E25`, representing 9.92 x 10^{25} (10 raised to the 25th power) or `5.7E-14`, representing 5.7 x 10^{-14} (10 raised to the -14th power). Both the mantissa (the number before the '`E`') and the exponent (the number after the '`E`') may be explicitly specified as positive (with a '`+`'), negative (with a '`-`') or unsigned (and therefore implicitly positive). A floating-point literals value must be within the range -1.7 x 10^{308} to $+1.7$ x 10^{308} with no more than 15 decimal digits of precision.

- Hexadecimal numeric literals
- Null terminated literals
- Raw C string using `L"characters"`.
- Binary using B#0 or 1.
- Octal using O#0 - 7. (That is the letter 'O').
- Hexadecimal number using `H#` or `X#` '0' - 'F'.
- Boolean Literals (Standard) `B" character "`.
- Boolean Literals (Hexadecimal) `BX" hex character "`.
- National Literals (Standard) `N" character "` or `NC" character "`.
- National Literals (Hexadecimal) `NX" character "`.

2.1.19.2. Alphanumeric Literals

An alphanumeric literal

An alphanumeric literal is not valid for use in arithmetic expressions unless it is first converted to its numeric computational equivalent; there are three numeric conversion intrinsic functions built into GnuCOBOL that can perform this conversion — `NUMVAL` (see [NUMVAL], page 496), `NUMVAL-C` (see [NUMVAL-C], page 497) and `NUMVAL-F` (see [NUMVAL-F], page 500).

Alphanumeric literals may take any of the following forms:

- A sequence of characters enclosed by a pair of single-quote (‘’)
- A literal formed according to the same rules as for a string literal (above), but prefixed with the letter 'Z' (upper- or lower-case) constitutes a zero-delimited string literal. These literals differ from ordinary string literals in that they will be explicitly terminated with a byte of hexadecimal value 00. These *Zero-Delimited Alphanumeric Literals*
- A *Hexadecimal Alphanumeric Literal*

Alphanumeric literals too long to fit on a single line may be continued to the next line in one of two ways:

1. If you are using Fixed Format Mode, the alphanumeric literal can be run right up to and including column 72. The literal may then be continued on the next line anywhere after column 11 by coding another quote or apostrophe (whichever was used to begin the literal originally). The continuation line must also have a hyphen (-)

```
          1         2         3         4         5         6         7
 12345678901234567890123456789012345678901234567890123456789012345678901234567890123

        01  LONG-LITERAL-VALUE-DEMO     PIC X(60) VALUE "This is a long l
       -                                             "ong literal that
       -                                             " must be continu
       -                                             "ed.".
```

2. Regardless of whether the compiler is operating in Fixed or Free Format Mode, GnuCOBOL allows alphanumeric literals to be broken up into separate fragments. These fragments have their own beginning and ending quote/apostrophe characters and are "glued together" at compilation time using '&'

```
         1         2         3         4         5         6         7
1234567890123456789012345678901234567890123456789012345678901234567890123

      01  LONG-LITERAL-VALUE-DEMO      PIC X(60) VALUE "This is a" &
                                        " long literal that must " &
                                                   "be continued.".
```

If your program is using Free Format Mode, there's less need to continue long alphanumeric literals because statements may be as long as 255 characters.

Numeric literals may be split across lines just as alphanumeric literals are, using either of the above techniques and both reserved and user-defined words can be split across lines too (using the first technique). The continuation of numeric literals and user-defined/reserved words is provided merely to provide compatibility with older COBOL versions and programs, but should not be used with new programs — it just makes for ugly-looking programs.

2.1.19.3. Figurative Constants

Figurative constants are reserved words that may be used as literals anywhere the figurative constants value could be interpreted as an arbitrarily long sequence of the characters in question. When a specific length is required, such as would be the case with an argument to a subprogram, a figurative constant may not be used. Thus, the following are valid uses of figurative constants:

```
05 FILLER                 PIC 9(10) VALUE ZEROS.
   ...
MOVE SPACES TO Employee-Name
```

But this is not:

```
CALL "SUBPGM" USING SPACES
```

The following are the GnuCOBOL figurative constants and their respective equivalent values.

`ZERO` This figurative constant has a value of numeric 0 (zero). `ZEROS` and `ZEROES` are both synonyms of `ZERO`.

`SPACE` This figurative constant has a value of one or more space characters. `SPACES` is a synonym of `SPACE`.

`QUOTE` This figurative constant has a value of one or more double-quote characters (`"`). `QUOTES` is a synonym of `QUOTE`.

`LOW-VALUE`
This figurative constant has a value of one or more of whatever character occupies the lowest position in the program's collating sequence as defined in the `OBJECT-COMPUTER` (see [OBJECT-COMPUTER], page 90) paragraph or — if no such specification was made — in whatever default character set the program is using (typically, this is the ASCII character set). `LOW-VALUES` is a synonym of `LOW-VALUE`.

When the character set in use is ASCII with no collating sequence modifications, the `LOW-VALUES` figurative constant value is the ASCII `NUL` character. Because character sets can be redefined, however, you should not rely on this fact. Use the `NULL` figurative constant instead.

`HIGH-VALUE`
: This figurative constant has a value of one or more of whatever character occupies the highest position in the program's collating sequence as defined in the `OBJECT-COMPUTER` paragraph or — if no such specification was made — in whatever default character set the program is using (typically, this is the ASCII character set). `HIGH-VALUES` is a synonym of `HIGH-VALUE`.

`NULL`
: A character comprised entirely of zero-bits (regardless of the programs collating sequence).

Programmers may create their own figurative constants via the `SYMBOLIC CHARACTERS` (see [Symbolic-Characters-Clause], page 100) clause of the `SPECIAL-NAMES` (see [SPECIAL-NAMES], page 92) paragraph.

2.1.20. Punctuation

A comma (‘`,`’)

The use of comma characters can cause confusion to a COBOL compiler if the `DECIMAL POINT IS COMMA` clause is used in the `SPECIAL-NAMES` (see [SPECIAL-NAMES], page 92) paragraph, as might be the case in Europe. The following statement, which calls a subroutine passing it two arguments (the numeric constants 1 and 2):

```
CALL "SUBROUTINE" USING 1,2
```

Would — with `DECIMAL POINT IS COMMA` in effect — actually be interpreted as a subroutine call with 1 argument (the non-integer numeric literal whose value is 1 and 2 tenths). For this reason, it is best to always follow a comma with a space.

The period character (‘`.`’)

The rules for where and when periods are needed in the procedure division are somewhat complicated. See [Use of Periods], page 54, for the details.

2.1.21. Interfacing to Other Environments

Through the `CALL` statement, COBOL programs may invoke other COBOL programs serving as subprograms. This is quite similar to cross-program linkage capabilities provided by other languages. In GnuCOBOL's case, the `CALL` facility is powerful enough to be tailored to the point where a GnuCOBOL program can communicate with operating system, database management and run-time library APIs, even if they weren't written in COBOL themselves. See [GnuCOBOL Main Programs CALLing C Subprograms], page 694, for an example of how a GnuCOBOL program could invoke a C-language subprogram, passing information back and forth between the two.

The fact that GnuCOBOL supports a full-featured two-way interface with C-language programs means that — even if you cannot access a library API directly — you could always do so via a small C “wrapper” program that is `CALL`ed by a GnuCOBOL program.

2.2. The COBOL Language - Advanced Techniques

2.2.1. Table References

COBOL uses parenthesis to specify the subscripts used to reference table entries (tables in COBOL are what other programming languages refer to as arrays).

For example, observe the following data structure which defines a 4 column by 3 row grid of characters:

```
01  GRID.
     05 GRID-ROW OCCURS 3 TIMES.
        10 GRID-COLUMN OCCURS 4 TIMES.
            15 GRID-CHARACTER       PIC X(1).
```

If the structure contains the following grid of characters:

```
A B C D
E F G H
I J K L
```

Then `GRID-CHARACTER (2, 3)` references the '`G`' and `GRID-CHARACTER (3, 2)` references the '`J`'.

Subscripts may be specified as numeric (integer) literals, numeric (integer) data items, data items created with any of the picture-less integer `USAGE` (see [USAGE], page 232) specifications, `USAGE INDEX` data items or arithmetic expressions resulting in a non-zero integer value.

In the above examples, a comma is used as a separator character between the two subscript values; semicolons (`;`) are also valid subscript separator characters, as are spaces! The use of a comma or semicolon separator in such a situation is technically optional, but by convention most COBOL programmers use one or the other. The use of no separator character (other than a space) is not recommended, even though it is syntactically correct, as this practice can lead to programmer-unfriendly code. It isn't too difficult to read and understand `GRID-CHARACTER(2 3)`, but it's another story entirely when trying to comprehend `GRID-CHARACTER(I + 1 J / 3)` (instead of `GRID-CHARACTER(I + 1, J / 3)`). The compiler accepts it, but too much of this would make *my* head hurt.

2.2.2. Qualification of Data Names

COBOL allows data names to be duplicated within a program, provided references to those data names may be made in such a manner as to make those references unique through a process known as qualification.

To see qualification at work, observe the following segments of two data records defined in a COBOL program:

```
01  EMPLOYEE.                        01  CUSTOMER.
    05 MAILING-ADDRESS.                  05 MAILING-ADDRESS.
       10 STREET        PIC X(35).          10 STREET        PIC X(35).
       10 CITY          PIC X(15).          10 CITY          PIC X(15).
       10 STATE         PIC X(2).           10 STATE         PIC X(2).
       10 ZIP-CODE.                         10 ZIP-CODE.
          15 ZIP-CODE-5 PIC 9(5).              15 ZIP-CODE-5 PIC 9(5).
          15 FILLER     PIC X(4).              15 FILLER     PIC X(4).
```

Now, let's deal with the problem of setting the `CITY` portion of an `EMPLOYEE`s `MAILING-ADDRESS` to '`Philadelphia`'. Clearly, `MOVE 'Philadelphia' TO CITY` cannot work because the compiler will be unable to determine which of the two `CITY` fields you are referring to.

In an attempt to correct the problem, we could qualify the reference to `CITY` as `MOVE 'Philadelphia' TO CITY OF MAILING-ADDRESS`.

Unfortunately that too is insufficient because it still insufficiently specifies which `CITY` is being referenced. To truly identify which specific `CITY` you want, you'd have to code `MOVE 'Philadelphia' TO CITY OF MAILING-ADDRESS OF EMPLOYEE`.

Now there can be no confusion as to which `CITY` is being changed. Fortunately, you don't need to be quite so specific; COBOL allows intermediate and unnecessary qualification levels to be omitted. This allows `MOVE 'Philadelphia' TO CITY OF EMPLOYEE` to do the job nicely.

If you need to qualify a reference to a table, do so by coding something like *`identifier-1`* `OF` *`identifier-2`* `(` *`subscript(s)`* `)`.

The reserved word `IN` may be used in lieu of `OF`.

2.2.3. Reference Modifiers

Reference Modifier (Format 1) Syntax

```
identifier-1 [ OF|IN identifier-2 ] [ (subscript...)  ]  (start:[ length ])
               ~~ ~~
```

Reference Modifier (Format 2) Syntax

```
intrinsic-function-reference (start:[ length ])
```

The COBOL 1985 standard introduced the concept of a reference modifier to facilitate references to only a portion of a data item; GnuCOBOL fully supports reference modification.

The *start* value indicates the starting character position being referenced (character position values start with 1, not 0 as is the case in some programming languages) and *length* specifies how many characters are wanted.

If no *length* is specified, a value equivalent to the remaining character positions from *start* to the end of *identifier-1* or to the end of the value returned by the function will be assumed.

Both *start* and *length* may be specified as integer numeric literals, integer numeric data items or arithmetic expressions with an integer value.

Here are a few examples:

`CUSTOMER-LAST-NAME (1:3)`
: References the first three characters of `CUSTOMER-LAST-NAME`.

`CUSTOMER-LAST-NAME (4:)`
: References all character positions of `CUSTOMER-LAST-NAME` from the fourth onward.

`FUNCTION CURRENT-DATE (5:2)`
: References the current month as a 2-digit number in character form. See [CURRENT-DATE], page 442, for more information.

`Hex-Digits (Nibble + 1:1)`
: Assuming that `Nibble` is a numeric data item with a value in the range 0-15, and Hex-Digits is a `PIC X(16)` item with a value of `0123456789ABCDEF`, this converts that numeric value to a hexadecimal digit.

`Table-Entry (6) (7:5)`
: References characters 7 through 11 (5 characters in total) in the 6th occurrence of Table-Entry.

Reference modification may be used anywhere an identifier is legal, including serving as the receiving field of statements like `MOVE` (see [MOVE], page 352), `STRING` (see [STRING], page 401) and `ACCEPT` (see [ACCEPT], page 268), to name a few.

2.2.4. Arithmetic Expressions

Arithmetic-Expression Syntax

```
Unary-Expression-1 { **|^ } Unary-Expression-2
                   {  *|/ }
                   {  +|- }
```

Unary-Expression Syntax

```
{ [ +|- ] { ( Arithmetic-Expression-1 )          } }
{         { [ LENGTH OF ] { identifier-1       } } }
{         {   ~~~~~~ ~~   { literal-1          } } }
{         {               { Function-Reference } } }
{ Arithmetic-Expression-2                          }
```

Arithmetic expressions are formed using four categories of operations — exponentiation, multiplication & division, addition & subtraction, and sign specification.

In complex expressions composed of multiple operators and operands, a precedence of operation applies whereby those operations having a higher precedence are computed first before operations with a lower precedence.

As is the case in almost any other programming language, the programmer is always free to use pairs of parenthesis to enclose sub-expressions of complex expressions that are to be evaluated before other sub-expressions rather than let operator precedence dictate the sequence of evaluation.

In highest to lowest order of precedence, here is a discussion of each category of operation:

Level 1 (Highest) — Unary Sign Specification (`+` and `-` with a single argument)
: The unary "minus" (-) operator returns the arithmetic negation of its single argument, effectively returning as its value the product of its argument and -1.

  The unary "plus" (`+`) operator returns the value of its single argument, effectively returning as its value the product of its argument and +1.

Level 2 — Exponentiation (`**` or `^`)
: The value of the left argument is raised to the power indicated by the right argument. Non-integer powers are allowed. The `^` and `**` operators are both supported to provide compatibility with programs written for other COBOL implementations.

Level 3 — Multiplication (`*`) and division (`/`)
: The `*` operator computes the product of the left and right arguments while the `/` operator computes the value of the left argument divided by the value of the right argument. If the right argument has a value of zero, expression evaluation

will be prematurely terminated before a value is generated. This may cause program failure at run-time.

A sequence of multiple 3rd-level operations (`A * B / C`, for example) will evaluate in strict left-to-right sequence if no parenthesis are used to control the order of evaluation.

Level 4 — Addition (`+`) or subtraction (`-`)

The `+` operator calculates the sum of the left and right arguments while the `-` operator computes the value of the right argument subtracted from that of the left argument.

A sequence of multiple 4th-level operations (`A - B + C`, for example) will evaluate in strict left-to-right sequence if no parenthesis are used to control the order of evaluation.

The syntactical rules of COBOL, allowing a dash (-) character in data item names, can lead to some ambiguity.

```
01  C         PIC 9 VALUE 5.
01  D         PIC 9 VALUE 2.
01  C-D       PIC 9 VALUE 7.
01  I         PIC 9 VALUE 0.
...
COMPUTE I=C-D+1
```

The `COMPUTE` (see [COMPUTE], page 301) statement will evaluate the arithmetic expression `C-D+1` and then save that result in `I`.

What value will be stored in `I`? The number 4, which is the result of subtracting the value of `D` (2) from the value of `C` (5) and then adding 1? Or, will it be the number 8, which is the value of adding 1 to the value of data item `C-D` (7)?

The right answer is 8 — the value of data item `C-D` plus 1! Hopefully, that was the intended result.

The GnuCOBOL compiler actually went through the following decision-making logic when generating code for the `COMPUTE` Statement:

1. Is there a data item named `C-D` defined? If so, use its value for the character sequence `C-D`.
2. If there is no `C-D` data item, then are there `C` and `D` data items? If not, the `COMPUTE` statement is in error. If there are, however, then code will be generated to subtract the value of `D` from `C` and add 1 to the result.

Had there been at least one space to the left and/or the right of the `-`, there would have been no ambiguity — the compiler would have been forced to use the individual `C` and `D` data items.

To avoid any possible ambiguity, as well as to improve program readability, it's considered good COBOL programming practice to always code at least one space to both the left and right of every operator in arithmetic expressions as well as the `=` sign on a COMPUTE.

Here are some examples of how the precedence of operations affects the results of arithmetic expressions (all examples use numeric literals, to simplify the discussion).

Expression	**Result**	**Notes**
3 * 4 + 1	13	* has precedence over +
4 * 2 ^ 3 - 10	22	2^3 is 8 (^ has precedence over *), times 4 is 32, minus 10 is 22.
(4 * 2) ^ 3 - 10	502	Parenthesis provide for a recursive application of the arithmetic expression rules, effectively allowing you to alter the precedence of operations. 4 times 2 is 8 (the use of parenthesis "trumps" the exponentiation operator, so the multiplication happens first); 8 ^ 3 is 512, minus 10 is 502.
5 / 2.5 + 7 * 2 - 1.15	14.85	Integer and non-integer operands may be freely intermixed

Of course, arithmetic expression operands may be numeric data items (any USAGE except POINTER or PROGRAM POINTER) as well as numeric literals.

2.2.5. Conditional Expressions

Conditional expressions are expressions which identify the circumstances under which a program may take an action or cease taking an action. As such, conditional expressions produce a value of `TRUE` or `FALSE`.

There are seven types of conditional expressions, as discussed in the following sections.

2.2.5.1. Condition Names

These are the simplest of all conditions. Observe the following code:

```
05  SHIRT-SIZE              PIC 99V9.
    88 TINY                 VALUE 0 THRU 12.5
    88 XS                   VALUE 13 THRU 13.5.
    88 S                    VALUE 14, 14.5.
    88 M                    VALUE 15, 15.5.
    88 L                    VALUE 16, 16.5.
    88 XL                   VALUE 17, 17.5.
    88 XXL                  VALUE 18, 18.5.
    88 XXXL                 VALUE 19, 19.5.
    88 VERY-LARGE           VALUE 20 THRU 99.9.
```

The condition names `TINY`, `XS`, `S`, `M`, `L`, `XL`, `XXL`, `XXXL` and `VERY-LARGE` will have `TRUE` or `FALSE` values based upon the values within their parent data item (`SHIRT-SIZE`).

A program wanting to test whether or not the current `SHIRT-SIZE` value can be classified as `XL` could have that decision coded as a combined condition (the most complex type of conditional expression), as either:

```
IF SHIRT-SIZE = 17 OR SHIRT-SIZE = 17.5

- or -

IF SHIRT-SIZE = 17 OR 17.5
```

Or it could simply utilize the condition name XL as follows:

```
IF XL
```

2.2.5.2. Class Conditions

Class-Condition Syntax

```
identifier-1 IS [ NOT ] { NUMERIC          }
                  ~~~   { ~~~~~~~          }
                        { ALPHABETIC       }
                        { ~~~~~~~~~~       }
                        { ALPHABETIC-LOWER }
                        { ~~~~~~~~~~~~~~~~ }
                        { ALPHABETIC-UPPER }
                        { ~~~~~~~~~~~~~~~~ }
                        { OMITTED          }
                        { ~~~~~~~          }
                        { class-name-1     }
```

Class conditions evaluate the type of data that is currently stored in a data item.

1. The `NUMERIC` class test considers only the characters '`0`', '`1`', . . . , '`9`' to be numeric; only a data item containing nothing but digits will pass a `NUMERIC` class test. Spaces, decimal points, commas, currency signs, plus signs, minus signs and any other characters except the digit characters will all fail `NUMERIC` class tests.
2. The `ALPHABETIC` class test considers only upper-case letters, lower-case letters and spaces to be alphabetic in nature.
3. The `ALPHABETIC-LOWER` and `ALPHABETIC-UPPER` class conditions consider only spaces and the respective type of letters to be acceptable in order to pass such a class test.
4. The `NOT` option reverses the `TRUE`/`FALSE` value of the condition.
5. Note that what constitutes a "letter" (or upper/lower case too, for that manner) may be influenced through the use of `CHARACTER CLASSIFICATION` specifications in the `OBJECT-COMPUTER` (see [OBJECT-COMPUTER], page 90) paragraph.
6. Only data items whose `USAGE` (see [USAGE], page 232) is either explicitly or implicitly defined as `DISPLAY` may be used in `NUMERIC` or any of the `ALPHABETIC` class conditions.
7. Some COBOL implementations disallow the use of group items or `PIC A` items with `NUMERIC` class conditions and the use of `PIC 9` items with `ALPHABETIC` class conditions. GnuCOBOL has no such restrictions.
8. The `OMITTED` class condition is used when it is necessary for a subprogram to determine whether or not a particular argument was passed to it. In such class conditions, *identifier-1* must be a linkage section item defined on the `USING` clause of the subprograms `PROCEDURE DIVISION` header. See [PROCEDURE DIVISION USING], page 248, for additional information.

The *class-name-1* option allows you to test for a user-defined class. Here's an example. First, assume the following `SPECIAL-NAMES` (see [SPECIAL-NAMES], page 92) definition of the user-defined class '`Hexadecimal`':

```
SPECIAL-NAMES.
    CLASS Hexadecimal IS '0' THRU '9', 'A' THRU 'F', 'a' THRU 'f'.
```

Now observe the following code, which will execute the `150-Process-Hex-Value` procedure if `Entered-Value` contains nothing but valid hexadecimal digits:

```
IF Entered-Value IS Hexadecimal
    PERFORM 150-Process-Hex-Value
END-IF
```

2.2.5.3. Sign Conditions

Sign-Condition Syntax

```
identifier-1 IS [ NOT ] { POSITIVE }
                  ~~~   { ~~~~~~~~ }
                        { NEGATIVE }
                        { ~~~~~~~~ }
                        { ZERO     }
                          ~~~~
```

Sign conditions evaluate the numeric state of a data item defined with a `PICTURE` (see [PICTURE], page 194) and/or `USAGE` (see [USAGE], page 232) that supports numeric values.

1. A `POSITIVE` or `NEGATIVE` class condition will be `TRUE` only if the value of *identifier-1* is strictly greater than or less than zero, respectively.
2. A `ZERO` class condition can be passed only if the value of *identifier-1* is exactly zero.
3. The `NOT` option reverses the `TRUE`/`FALSE` value of the condition.

2.2.5.4. Switch-Status Conditions

In the `SPECIAL-NAMES` paragraph, an external switch name can be associated with one or more condition names. These condition names may then be used to test the ON/OFF status of the external switch.

Here are the relevant sections of code in a program named `testprog`, which is designed to simply announce if `SWITCH-1` is on:

```
...
ENVIRONMENT DIVISION.
SPECIAL-NAMES.
    SWITCH-1 ON STATUS IS Switch-1-Is-ON.
...
PROCEDURE DIVISION.
...
    IF Switch-1-Is-ON
        DISPLAY "Switch 1 Is On"
    END-IF
...
```

The following are two different command window sessions — the left on a Unix/Cygwin/OSX system and the right on a windows system — that will set the switch on and then execute the `testprog` program. Notice how the message indicating that the program detected the switch was set is displayed in both examples:

```
$ COB_SWITCH_1=ON           C:>SET COB_SWITCH_1=ON
$ export COB_SWITCH_1       C:>testprog
$ ./testprog                Switch 1 Is On
Switch 1 Is On              C:>
$
```

2.2.5.5. Relation Conditions

Relation-Condition Syntax

```
{ identifier-1            } IS [ NOT ] RelOp { identifier-2            }
{ literal-1               }      ~~~         { literal-2               }
{ arithmetic-expression-1 }                  { arithmetic-expression-2 }
{ index-name-1            }                  { index-name-2            }
```

RelOp Syntax

```
{ EQUAL TO                 }
{ ~~~~~                    }
{ EQUALS                   }
{ ~~~~~~                   }
{ GREATER THAN             }
{ ~~~~~~~                  }
{ GREATER THAN OR EQUAL TO }
{ ~~~~~~~      ~~ ~~~~~    }
{ LESS THAN                }
{ ~~~~                     }
{ LESS THAN OR EQUAL TO    }
{ ~~~~      ~~ ~~~~~       }
{ =                        }
{ >                        }
{ >=                       }
{ <                        }
{ <=                       }
```

These conditions evaluate how two different values "relate" to each other.

1. When comparing one numeric value to another, the `USAGE` (see [USAGE], page 232) and number of significant digits in either value are irrelevant as the comparison is performed using the actual algebraic values.
2. When comparing strings, the comparison is made based upon the program's collating sequence. When the two string arguments are of unequal length, the shorter is assumed to be padded (on the right) with a sufficient number of spaces as to make the two strings of equal length. String comparisons take place on a corresponding character-by-character basis, left to right, until the `TRUE/FALSE` value for the relation test can be established. Characters are compared according to their relative position in the program's `COLLATING SEQUENCE` (as defined in `SPECIAL-NAMES` (see [SPECIAL-NAMES], page 92)), *not* according to the bit-pattern values the characters have in storage.

3. By default, the program's `COLLATING SEQUENCE` will, however, be based entirely on the bit-pattern values of the various characters.
4. There is no functional difference between using the wordy version (`IS EQUAL TO`, `IS LESS THAN`, . . .) versus the symbolic version (`=`, `<`, . . .) of the actual relation operators.

2.2.5.6. Combined Conditions

Combined Condition Syntax

```
[ ( ] Condition-1 [ ) ] { AND } [ ( ] Condition-2 [ ) ]
                       { ~~~ }
                       { OR  }
                       { ~~  }
```

A combined condition is one that computes a TRUE/FALSE value from the TRUE/FALSE values of two other conditions (which could themselves be combined conditions).

1. If either condition has a value of TRUE, the result of ORing the two together will result in a value of TRUE. ORing two FALSE conditions will result in a value of FALSE.
2. In order for AND to yield a value of TRUE, both conditions must have a value of TRUE. In all other circumstances, AND produces a FALSE value.
3. When chaining multiple, similar conditions together with the same operator (OR/AND), and left or right arguments have common subjects, it is possible to abbreviate the program code. For example:

   ```
   IF ACCOUNT-STATUS = 1 OR ACCOUNT-STATUS = 2 OR ACCOUNT-STATUS = 7
   ```

   Could be abbreviated as:

   ```
   IF ACCOUNT-STATUS = 1 OR 2 OR 7
   ```

4. Just as multiplication takes precedence over addition in arithmetic expressions, so does AND take precedence over OR in combined conditions. Use parenthesis to change this precedence, if necessary. For example:

   `FALSE AND FALSE OR TRUE AND TRUE`
   : Evaluates to TRUE

   `(FALSE AND FALSE) OR (TRUE AND TRUE)`
   : Evaluates to TRUE (since AND has precedence over OR) - this is identical to the previous example

   `(FALSE AND (FALSE OR TRUE)) AND TRUE`
   : Evaluates to FALSE

2.2.5.7. Negated Conditions

Negated Condition Syntax

```
NOT Condition-1
~~~
```

A condition may be negated by prefixing it with the `NOT` operator.

1. The `NOT` operator has the highest precedence of all logical operators, just as a unary minus sign (which "negates" a numeric value) is the highest precedence arithmetic operator.
2. Parenthesis must be used to explicitly signify the sequence in which conditions are evaluated and processed if the default precedence isn't desired. For example:

   `NOT TRUE AND FALSE AND NOT FALSE`
   : Evaluates to `FALSE AND FALSE AND TRUE` which evaluates to `FALSE`

   `NOT (TRUE AND FALSE AND NOT FALSE)`
   : Evaluates to `NOT (FALSE)` which evaluates to `TRUE`

   `NOT TRUE AND (FALSE AND NOT FALSE)`
   : Evaluates to `FALSE AND (FALSE AND TRUE)` which evaluates to `FALSE`

2.2.6. Use of Periods

All COBOL implementations distinguish between sentences and statements in the procedure division. A *Statement* is a single executable COBOL instruction. For example, these are all statements:

```
MOVE SPACES TO Employee-Address
ADD 1 TO Record-Counter
DISPLAY "Record-Counter=" Record-Counter
```

Some COBOL statements have a *scope of applicability* associated with them where one or more other statements can be considered to be part of or related to the statement in question. An example of such a situation might be the following, where the interest on a loan is being calculated and displayed at 4% interest if the loan balance is under $10,000, and 4.5% otherwise. (WARNING: the following code has an error!):

```
IF Loan-Balance < 10000
    MULTIPLY Loan-Balance BY 0.04 GIVING Interest
ELSE
    MULTIPLY Loan-Balance BY 0.045 GIVING Interest
DISPLAY "Interest Amount = " Interest
```

In this example, the `IF` statement actually has a scope that can include two sets of associated statements: one set to be executed when the `IF` (see [IF], page 338) condition is `TRUE`, and another if it is `FALSE`.

Unfortunately, there's a problem with the above. A human being looking at that code would probably infer that the `DISPLAY` (see [DISPLAY], page 305) statement, because of its lack of indentation, is to be executed regardless of the `TRUE`/`FALSE` value of the `IF` condition. Unfortunately, the compiler (any COBOL compiler) won't see it that way because it really couldn't care less what sort of indentation, if any, is used. In fact, any COBOL compiler would be just as happy to see the code written like this:

```
IF Loan-Balance < 10000 MULTIPLY Loan-balance
BY 0.04 GIVING Interest ELSE MULTIPLY
Loan-Balance BY 0.045 GIVING Interest DISPLAY
"Interest Amount = " Interest
```

How then do we inform the compiler that the `DISPLAY` statement is outside the scope of the `IF`?

That's where sentences come in.

A COBOL *Sentence* is defined as any arbitrarily long sequence of statements, followed by a period (.) character. The period character is what terminates the scope of a set of statements. Therefore, our example should have been coded like this:

```
IF Loan-Balance < 10000
    MULTIPLY Loan-Balance BY 0.04 GIVING Interest
ELSE
    MULTIPLY Loan-Balance BY 0.045 GIVING Interest.
DISPLAY "Interest Amount = " Interest
```

See the period at the end of the second `MULTIPLY` (see [MULTIPLY], page 354)? That is what terminates the scope of the `IF`, thus making the `DISPLAY` statement's execution completely independent of the `TRUE`/`FALSE` status of the `IF`.

2.2.7. Use of VERB/END-VERB Constructs

Prior to the 1985 COBOL standard, using a period character was the only way to signal the end of a statement's scope.

Unfortunately, this caused some problems. Take a look at this code:

```
IF A = 1
    IF B = 1
        DISPLAY "A & B = 1"
ELSE *> This ELSE has a problem!
    IF B = 1
        DISPLAY "A NOT = 1 BUT B = 1"
    ELSE
        DISPLAY "NEITHER A NOR B = 1".
```

The problem with this code is that indentation — so critical to improving the human-readability of a program — can provide an erroneous view of the logical flow. An `ELSE` is always associated with the most-recently encountered `IF`; this means the emphasized `ELSE` will be associated with the `IF B = 1` statement, not the `IF A = 1` statement as the indentation would appear to imply.

This sort of problem led to a band-aid solution being added to the COBOL language: the `NEXT SENTENCE` clause:

```
IF A = 1
    IF B = 1
        DISPLAY "A & B = 1"
    ELSE
        NEXT SENTENCE
ELSE
    IF B = 1
        DISPLAY "A NOT = 1 BUT B = 1"
    ELSE
        DISPLAY "NEITHER A NOR B = 1".
```

`NEXT SENTENCE` informs the compiler that if the `B = 1` condition is false, control should fall into the first statement that follows the next period.

With the 1985 standard for COBOL, a much more elegant solution was introduced. Any COBOL *Verb* (the first reserved word of a statement) that needed such a thing was allowed to use an `END-`*`verb`* construct to end its scope without disrupting the scope of any other statement it might have been in. Any COBOL 85 compiler would have allowed the following solution to our problem:

```
IF A = 1
    IF B = 1
        DISPLAY "A & B = 1"
    END-IF
ELSE
    IF B = 1
        DISPLAY "A NOT = 1 BUT B = 1"
    ELSE
```

```
            DISPLAY "NEITHER A NOR B = 1".
```

This new facility made the period almost obsolete, as our program segment would probably be coded like this today:

```
IF A = 1
    IF B = 1
        DISPLAY "A & B = 1"
    END-IF
ELSE
    IF B = 1
        DISPLAY "A NOT = 1 BUT B = 1"
    ELSE
        DISPLAY "NEITHER A NOR B = 1"
    END-IF
END-IF
```

COBOL (GnuCOBOL included) still requires that each procedure division paragraph contain at least one sentence if there is any executable code in that paragraph, but a popular coding style is now to simply code a single period right before the end of each paragraph.

The standard for the COBOL language shows the various `END-`*`verb`* clauses are optional because using a period as a scope-terminator remains legal.

If you will be porting existing code over to GnuCOBOL, you'll find it an accommodating facility capable of conforming to whatever language and coding standards that code is likely to use. If you are creating new GnuCOBOL programs, however, I would strongly counsel you to use the `END-`*`verb`* structures in those programs.

2.2.8. Concurrent Access to Files

The manipulation of data files is one of the COBOL language's great strengths. There are features built into COBOL to deal with the possibility that multiple programs may be attempting to access the same file concurrently. Multiple program concurrent access is dealt with in two ways — file sharing and record locking.

Not all GnuCOBOL implementations support file sharing and record-locking options. Whether they do or not depends upon the operating system they were built for and the build options that were used when the specific GnuCOBOL implementation was generated.

2.2.8.1. File Sharing

GnuCOBOL controls concurrent-file access at the highest level through the concept of file sharing, enforced when a program attempts to open a file. This is accomplished via a UNIX operating-system routine called `fcntl`. That module is not currently supported by Windows and is not present in the MinGW Unix-emulation package. GnuCOBOL builds created using a MinGW environment will be incapable of supporting file-sharing controls — files will always be shared in such environments. A GnuCOBOL build created using the Cygwin environment on Windows *would* have access to `fcntl` and therefore *will* support file sharing. Of course, actual Unix builds of GnuCOBOL, as well as OSX builds, should have no issues because `fcntl` should be available.

Any limitations imposed on a successful `OPEN` (see [OPEN], page 358) will remain in place until your program either issues a `CLOSE` (see [CLOSE], page 299) against the file or the program terminates.

File sharing is controlled through the use of a `SHARING` clause:

```
SHARING WITH { ALL OTHER }
~~~~~~~      { ~~~       }
             { NO OTHER  }
             { ~~        }
             { READ ONLY }
               ~~~~ ~~~~
```

This clause may be used either in the file's `SELECT` statement (see [SELECT], page 104), on the `OPEN` statement (see [OPEN], page 358) which initiates your program's use of the file, or both. If a `SHARING` option is specified in *both* places, the specifications made on the `OPEN` statement will take precedence over those from the `SELECT` statement.

Here are the meanings of the three options:

`ALL OTHER` When your program opens a file with this sharing option in effect, no restrictions will be placed on other programs attempting to `OPEN` the file after your program did. This is the default sharing mode.

`NO OTHER` When your program opens a file with this sharing option in effect, your program announces that it is unwilling to allow any other program to have any access to the file as long as you are using that file; `OPEN` attempts made in other programs will fail with a file status of 37 (`PERMISSION DENIED`) until such time as you `CLOSE` (see [CLOSE], page 299) the file.

`READ ONLY` Opening a file with this sharing option indicates you are willing to allow other programs to `OPEN` the file for input while you have it open. If they attempt any other `OPEN`, theirs will fail with a file status of 37. Of course, your program may fail if someone else got to the file first and opened it with a sharing option that imposed file-sharing limitations.

If the `SELECT` of a file is coded with a `FILE STATUS` clause, `OPEN` failures — including those induced by sharing failures — will be detectable by the program and a graceful recovery (or at least a graceful termination) will be possible. If no such clause was coded, however, a runtime message will be issued and the program will be terminated.

2.2.8.2. Record Locking

Record-locking is supported by advanced file-management software built-in to the GnuCOBOL implementation you are using. This software provides a single point-of-control for access to files — usually `ORGANIZATION INDEXED` files. One such runtime package capable of doing this is the Berkeley Database (BDB) package — a package frequently used in GnuCOBOL builds to support indexed files.

The various I/O statements your program can execute are capable of imposing limitations on access by other concurrently-executing programs to the file record they just accessed. These limitations are syntactically imposed by placing a lock on the record using a `LOCK` clause. Other records in the file remain available, assuming that file-sharing limitations imposed at the time the file was opened didn't prevent access to the entire file.

1. If the GnuCOBOL build you are using was configured to use the Berkeley Database (BDB) package for indexed file I/O, record locking will be available by using the `DB_HOME` run-time environment variable.
2. If the `SELECT` (see [SELECT], page 104) statement or file `OPEN` (see [OPEN], page 358) specifies `SHARING WITH NO OTHER`, record locking will be disabled.
3. If the file's `SELECT` contains a `LOCK MODE IS AUTOMATIC` clause, every time a record is read from the file, that record is automatically locked. Other programs may access *other* records within the file, but not a locked record.
4. If the file's `SELECT` contains a `LOCK MODE IS MANUAL` clause, locks are placed on records *only* when a `READ` statement executed against the file includes a `LOCK` clause (this clause will be discussed shortly).
5. If the `LOCK ON` clause *is* specified in the file's `SELECT`, locks (either automatically or manually acquired) will continue to accumulate as more and more records are read, until they are explicitly released. This is referred to as *multiple record locking*.

   Locks acquired via multiple record locking remain in effect until the program holding the lock. . .

   - . . . terminates, or . . .
   - . . . executes a `CLOSE` statement (see [CLOSE], page 299) against the file, or . . .
   - . . . executes an `UNLOCK` statement (see [UNLOCK], page 412) against the file, or . . .
   - . . . executes a `COMMIT` statement (see [COMMIT], page 300) or . . .
   - . . . executes a `ROLLBACK` statement (see [ROLLBACK], page 376).
6. If the `LOCK ON` clause is *not* specified, then the next I/O statement your program executes, except for `START` (see [START], page 397), will release the lock. This is referred to as *single record locking*.
7. A `LOCK` clause, which may be coded on a `READ` (see [READ], page 366), `REWRITE` (see [REWRITE], page 374) or `WRITE` statement (see [WRITE], page 417) looks like this:

```
{ IGNORING LOCK    }
{ ~~~~~~~~ ~~~~    }
{ WITH [ NO ] LOCK }
{        ~~   ~~~~ }
```

```
{ WITH KEPT LOCK   }
{      ~~~~ ~~~~   }
{ WITH IGNORE LOCK }
{      ~~~~~~ ~~~~ }
{ WITH WAIT        }
       ~~~~
```

The `WITH [ NO ] LOCK` option is the only one available to `REWRITE` or `WRITE` statements.

The meanings of the various record locking options are as follows:

`IGNORING LOCK`
WITH IGNORE LOCK
: These options (which are synonymous) inform GnuCOBOL that any locks held by other programs should be ignored.

`WITH LOCK`
: Access to the record by other programs will be denied.

`WITH NO LOCK`
: The record will not be locked. This is the default for all statements.

`WITH KEPT LOCK`
: When single record locking is in effect, as a new record is accessed, locks held for previous records are released. By using this option, not only is the newly accessed record locked (as `WITH LOCK` would do), but prior record locks will be retained as well. A subsequent `READ` without the `KEPT LOCK` option will release all "kept" locks, as will the `UNLOCK` statement.

`WITH WAIT`
: This option informs GnuCOBOL that the program is willing to wait for a lock held (by another program) on the record being read to be released.

  Without this option, an attempt to read a locked record will be immediately aborted and a file status of 51 will be returned.

  With this option, the program will wait for a preconfigured time for the lock to be released. If the lock is released within the preconfigured wait time, the read will be successful. If the preconfigured wait time expires before the lock is released, the read attempt will be aborted and a 51 file status will be issued.

End of Chapter 2 — COBOL Fundamentals

3. CDF - Compiler Directing Facility

The Compiler Directing Facility, or CDF, is a means of controlling the compilation of GnuCOBOL programs. CDF provides a mechanism for dynamically setting or resetting certain compiler switches, introducing new source code from one or more source code libraries, making dynamic source code modifications and conditionally processing or ignoring source statements altogether. This is accomplished via a series of special CDF statements and directives that will appear in the program source code.

When the compiler is operating in Fixed Format Mode, all CDF statements must begin in column seven (7) or beyond.

There are two types of supported CDF statements in GnuCOBOL — Text Manipulation Statements and Compiler Directives.

The CDF text manipulation statements `COPY` and `REPLACE` are used to introduce new code into programs either with or without changes, or may be used to modify existing statements already in the program. Text manipulation statements are always terminated with a period.

CDF directives, denoted by the presence of a `>>` character sequence as part of the statement name itself, influence the process of program compilation.

Compiler directives are *never* terminated with a period.

The compiler command-line option `-D` offers additional control (see ⟨undefined⟩ [cobc - The GnuCOBOL Compiler], page ⟨undefined⟩).

3.1. >>CALL-CONVENTION

CDF >>CALL-CONVENTION Syntax

```
>>CALL-CONVENTION    { COBOL   }
~~~~~~~~~~~~~~~~~    { EXTERN  }
                     { STDCALL }
                     { STATIC  }
```

This directive instructs the compiler how to treat references to program names and may be used to determine other details for interacting with a function or program. There are four options with COBOL being the default.

`COBOL` The program name is treated as a COBOL word that maps to the externalised name program to be called, cancelled or referenced in the program-address-identifier, applying the same mapping rules as for a program name for which no `AS` phrase is specified. (The is the default.)

`EXTERN` The program name is treated as an external reference.

`STDCALL` [more info needed]

`STATIC` The program name is called as a included element and not dynamically which is the normal default.

3.2. COPY

CDF COPY Statement Syntax

```
COPY copybook-name
~~~~
[ IN|OF library-name ]
  ~~ ~~
[ SUPPRESS PRINTING ]
  ~~~~~~~~
[ REPLACING { Phrase-Clause | String-Clause }...  ]  .
  ~~~~~~~~~
```

CDF COPY Phrase-Clause Syntax

```
{ ==pseudo-text-1== } BY { ==pseudo-text-2== }
{ identifier-1      } ~~ { identifier-2      }
{ literal-1         }    { literal-2         }
{ word-1            }    { word-2            }
```

CDF COPY String-Clause Syntax

```
[ LEADING|TRAILING ] ==partial-word-1== BY ==partial-word-2==
  ~~~~~~~ ~~~~~~~~                        ~~
```

---

1. `COPY` statements are used to import copybooks (see [Copybooks], page 14) into a program.
2. `COPY` statements may be used anywhere within a COBOL program where the code contained within the copybook would be syntactically valid.
3. The optional `SUPPRESS` clause (with or without the optional `PRINTING` reserved word) is valid syntactically but is non-functional. It is supported to facilitate compatibility with source code written for other versions of COBOL.
4. There is no difference between the use of the word `IN` and the word `OF` — use the one you prefer.
5. A period is absolutely mandatory at the end of every `COPY` statement, even if the statement occurs within the scope of another one where a period might appear disruptive, such as within the scope of an `IF` (see [IF], page 338) statement. This mandatory period at the end of the statement does not, however, affect the statement scope in which the `COPY` occurs.

6. Both *pseudo-text-2* and *partial-word-2* may be null.
7. All `COPY` statements are located and the contents of the corresponding copybooks inserted into the program source code before the actual compilation process begins. If a copybook contains a `COPY` statement, the copybook insertion process will be repeated to resolve the embedded `COPY`. This will continue until no unresolved `COPY` statements remain. At that point, actual program compilation will begin.
8. See [Locating Copybooks], page 648, for the specific rules on how copybooks are located by the compiler.
9. The optional `REPLACING` clause allows for one or more of either of the following kinds of text replacements to be made:

   Phrase-Clause

   Replacement of one or more complete reserved words, user-defined identifiers or literals; the following points apply to this option:

   - This option cannot be used to replace part of a word, identifier or literal.
   - Whatever precedes the `BY` will be referred to here as the search string.
   - Single-item search strings can be specified by coding the *`identifier-1`*, *`literal-1`* or *`word-1`* being replaced.
   - Multiple-item search strings can be specified using the `==`*`pseudo-text-1`*`==` option. For example, to replace all occurrences of `UPON PRINTER`, you would specify `==UPON PRINTER==`.
   - The replacement string, which follows the `BY`, may be specified using any of the four options.
   - If the replacement string is a multiple-item phrase or is to be deleted altogether, you must use the `==`*`pseudo-text-2`*`==` option. If *`pseudo-text-2`* is null (in other words, the replacement text is specified as `====`), all encountered occurrences of the search string will be deleted.

   String-Clause

   Using this, you may replace character sequences that occur at the beginning (see `LEADING`) or end (see `TRAILING`) of reserved or user-defined words. For example, to change all words of the form `"0100-xxxxxx"` to `"020-xxxxxx"`, code `LEADING ==0100-== BY ==020-==`. To simply remove all `"0100-"` prefixes from words, code `LEADING ==0100-== BY ====`.

3.3. REPLACE

CDF REPLACE Statement (Format 1) Syntax

```
REPLACE [ ALSO ] { Phrase-Clause | String-Clause }...  .
~~~~~~~   ~~~~
```

CDF REPLACE Statement (Format 2) Syntax

```
REPLACE [ LAST ] OFF .
~~~~~~~   ~~~~   ~~~
```

CDF REPLACE Phrase-Clause Syntax

```
{ ==pseudo-text-1== } BY { ==pseudo-text-2== }
                     ~~
```

CDF REPLACE String-Clause Syntax

```
[ LEADING|TRAILING ] ==partial-word-1== BY ==partial-word-2==
  ~~~~~~~ ~~~~~~~~                       ~~
```

---

1. The `REPLACE` statement provides a mechanism for changing all or part of one or more GnuCOBOL statements.
2. A period is absolutely mandatory at the end of every `REPLACE` statement (either format), even if the statement occurs within the scope of another one where a period might appear disruptive (such as within the scope of an `IF` (see [IF], page 338) statement; the period will not, however, affect the statement scope in which the `REPLACE` occurs.
3. The following points apply to Format 1 of the `REPLACE` statement:
   A. Format 1 of the `REPLACE` statement can be used to make changes to program source code in much the same way as the `REPLACING` option of the `COPY` statement can, via these options:

      Phrase-Clause

      Replace one or more complete reserved words, user-defined identifiers or literals; the following points apply to this option:

      - This option cannot be used to replace part of a word, identifier or literal.

- Whatever precedes the `BY` will be referred to here as the search string.
- Search strings on `REPLACE` are always specified using the `==pseudo-text-1==` option. For example, to replace all occurrences of `UPON PRINTER`, you would specify `==UPON PRINTER==`.
- The replacement string, which follows the `BY`, is specified using the `==pseudo-text-2==` option. If *pseudo-text-2* is null (in other words, the replacement text is specified as `====`), all encountered occurrences of the search string will be deleted.

String-Clause

Using this, you may replace character sequences that occur at the beginning (see `LEADING`) or end (see `TRAILING`) of reserved or user-defined words. For example, to change all words of the form `"0100-xxxxxx"` to `"020-xxxxxx"`, code `LEADING ==0100-== BY ==020-==`. To simply remove all `"0100-"` prefixes from words, code `LEADING ==0100-== BY ====`.

B. Once a Format 1 `REPLACE` statement is encountered in the currently-compiling source file, Replace Mode becomes active, and the change(s) specified by that statement will be automatically made on all subsequent source statements the compiler reads from the file.

C. Replace Mode remains in-effect — continuing to make source code changes — until another Format 1 `REPLACE` is encountered, the end of currently compiling program source file is reached or a Format 2 `REPLACE` statement is encountered.

D. When a Format 1 `REPLACE` statement with the `ALSO` keyword is encountered without Replace Mode being currently active, the effect will be as if the `ALSO` had not been specified. If Replace Mode already was in effect, the effect will be to "push" the current change specification(s) onto the top of a stack and add the specification(s) of the new statement to those that were already in effect.

E. When a Format 1 `REPLACE` without the `ALSO` keyword is encountered, any stacked change specification(s), if any, will be discarded and the currently in-effect change specification(s), if any, will be replaced by those of the new statement.

F. When the end of the currently-compiling source file is reached, Replace Mode is deactivated and any stacked replace specifications will be discarded — compilation of the next source file (if any) will begin with Replace Mode inactive and no change specification(s) on the stack.

4. The following points apply to Format 2 of the `REPLACE` statement:

A. If Replace Mode is currently inactive, the Format 2 `REPLACE` statement will be ignored.

B. If Replace Mode is currently active, a `REPLACE OFF.` will deactivate Replace Mode and discard any replace specification(s) on the stack. The compiler will henceforth operate as if no `REPLACE` had ever been encountered, until such time as another Format 1 `REPLACE` is encountered.

C. If Replace Mode is currently active, a `REPLACE LAST OFF.` will replace the current replace specification(s) with those popped off the top of the stack. If there were no replace specification(s) on the stack, the effect will be as if a `REPLACE OFF.` had been coded.

3.4. >>DEFINE

CDF >>DEFINE Directive Syntax

```
>>DEFINE [ CONSTANT ] cdf-variable-1 AS { OFF                    }
~~~~~~~~   ~~~~~~~~                     { ~~~                    }
                                        { literal-1 [ OVERRIDE ] }
                                        {             ~~~~~~~~   }
                                        { PARAMETER [ OVERRIDE ] }
                                          ~~~~~~~~~   ~~~~~~~~
```

Use the `>>DEFINE` CDF directive to create CDF variables and (optionally) assign them either literal or environment variable values.

1. The reserved word `AS` is optional and may be included, or not, at the discretion of the programmer. The presence or absence of this word has no effect upon the program.
2. CDF variables defined in this way become undefined once an `END PROGRAM` or `END FUNCTION` directive is encountered in the input source.
3. The `>>DEFINE` CDF directive is one way to create CDF variables that may be processed by other CDF statements such as `>>IF` (see [>>IF], page 69). The `>>SET` CDF directive (see [>>SET], page 71) provides another way to create them.
4. CDF variable names follow the rules for standard GnuCOBOL user-defined names, and may not duplicate any CDF reserved word. CDF variable names may duplicate COBOL reserved words, provided the `CONSTANT` option is not specified, but such names are not recommended.
5. The `CONSTANT` option is valid only in conjunction with *literal-1*. When `CONSTANT` is specified, the CDF variable that is created may be used within your regular COBOL code as if it were a literal value. Without this option, the CDF variable may only be referenced on other CDF statements. The `OFF` option is used to create a variable without assigning it any value.
6. The `PARAMETER` option is used to create a variable whose value is that of the environment variable of the same name. Note that this value assignment occurs at compilation time, not program execution time.
7. In the absence of the `OVERRIDE` option, *cdf-variable-1* must not yet have been defined. When the `OVERRIDE` option is specified, *cdf-variable-1* will be created with the specified value, if it had not yet been defined. If it had already been defined, it will be redefined with the new value.

3.5. >>IF

CDF >>IF Directive Syntax

```
>>IF CDF-Conditional-Expression-1
~~~~     [ Program-Source-Lines-1 ]

[ >>ELIF CDF-Conditional-Expression-2
  ~~~~~~ [ Program-Source-Lines-2 ] ]...

[ >>ELSE
  ~~~~~~ [ Program-Source-Lines-3 ] ]

>>END-IF
~~~~~~~~
```

CDF-Conditional-Expression Syntax

```
{ cdf-variable-1 } IS [ NOT ] { DEFINED                      }
{ literal-1      }      ~~~   { ~~~~~~~                      }
                              { SET                          }
                              { ~~~                          }
                              { CDF-RelOp { cdf-variable-2 } }
                              {           { literal-2      } }
```

CDF-RelOp Syntax

```
>=    or    GREATER THAN OR EQUAL TO
            ~~~~~~~      ~~ ~~~~~
>     or    GREATER THAN
            ~~~~~~~
<=    or    LESS THAN OR EQUAL TO
            ~~~~      ~~ ~~~~~
<     or    LESS THAN
            ~~~~
=     or    EQUAL TO
            ~~~~~
<>    or    EQUAL TO (with "NOT")
            ~~~~~
```

The >>IF CDF directive causes the GnuCOBOL compiler to process or ignore COBOL source statements, CDF text-manipulation statements and CDF directives depending upon the value of one or more conditional expressions based upon CDF variables.

1. The reserved words IS, THAN and TO are optional and may be omitted. The presence or absence of these words has no effect on the program.
2. Each >>IF directive must be terminated by an >>END-IF directive.
3. There may be any number of >>ELIF clauses following an >>IF, including zero.
4. There may no more than one >>ELSE clause following an >>IF. When >>ELSE is used, it must follow the >>IF and all >>ELIF clauses.
5. Only one of the *Program-Source-Lines-n* block of statements that lie within the scope of the >>IF ... >>END-IF may be processed by the compiler. Which one (if any) that gets processed will be decided as follows:
   A. Each *CDF-Conditional-Expression-n* will be evaluated, in turn, in the sequence in which they are coded in the >>IF statement and any >>ELIF clauses that may be present until one evaluates to TRUE. Once one of them evaluates to TRUE, the *Program-Source-Lines-n* block of code that corresponds to the TRUE *CDF-Conditional-Expression-n* will be one that is processed. All others within the >>IF->>END-IF scope will be ignored.
   B. If no *CDF-Conditional-Expression* evaluates to TRUE, and there is an >>ELSE clause, the *Program-Source-Lines-3* block of statements following the >>ELSE clause will be processed by the compiler and all others within the >>IF->>END-IF scope will be ignored.
   C. If no *CDF-Conditional-Expression-n* evaluates to TRUE and there is no >>ELSE clause, then none of the *Program-Source-Lines-n* block of statements within the >>IF->>END-IF scope will be processed by the compiler.
   D. If the *Program-Source-Lines-n*> statement block selected for processing is empty, no error results — there will just be no code generated from the >>IF->>END-IF structure.
6. A *Program-Source-Lines-n* block may contain any valid COBOL or CDF code.
7. The following points pertain to any *CDF-Conditional-Expression-n*:
   A. The DEFINED option tests for whether *cdf-variable-1* has been defined, but not yet assigned a value (>>DEFINE ... OFF); use the NOT option to test for the variable not being defined.
   B. The SET option tests for whether *cdf-variable-1* has been given a value, either via a >>SET statement or via a >>DEFINE without the OFF option.
   C. Two CDF variables, two literals or a single CDF variable and a single literal may be compared against each other using a relational operator. Unlike the standard GnuCOBOL IF statement (see [IF], page 338), multiple comparisons cannot be ANDed or ORed together; you may nest a second >>IF inside the first, however, to simulate an AND and an OR may be simulated via the >>ELIF option.
   D. The <> symbol stands for NOT EQUAL TO.

3.6. >>SET

CDF >>SET Directive Syntax

```
>>SET { [ CONSTANT ] cdf-variable-1 literal-1 ]                              }
~~~~~ {   ~~~~~~~~                                                          }
      { SOURCEFORMAT AS FIXED|FREE|VARIABLE|XOPEN|XCARD|CRT|TERMINAL|COBOLX }
      { ~~~~~~~~~~~~    ~~~~~ ~~~~ ~~~~~~~~ ~~~~~ ~~~~~ ~~~ ~~~~~~~~ ~~~~~~ }
      { NOFOLDCOPYNAME                                                      }
      { ~~~~~~~~~~~~~~                                                      }
      { FOLDCOPYNAME AS UPPER|LOWER                                         }
        ~~~~~~~~~~~~    ~~~~~ ~~~~~
```

The >>SET CDF directive provides an alternate means of performing the actions of the >>DEFINE and >>SOURCE directives, as well as a means of controlling the compiler's -free switch, -fixed switch and -ffold-copy switch from within program source code.

1. The reserved word AS is optional (only on the SOURCEFORMAT and FOLDCOPYNAME clauses) and may be included, or not, at the discretion of the programmer. The presence or absence of this word has no effect upon the program.
2. CDF variables defined in this way become undefined once an END PROGRAM or END FUNCTION directive is encountered in the input source.
3. The FOLDCOPYNAME option provides the equivalent of specifying the compiler -ffold-copy=*xxx* switch, where *xxx* is either UPPER or LOWER.
4. The NOFOLDCOPYNAME option turns off the effect of either the >>SET FOLDCOPYNAME statement or the compiler -ffold-copy=*xxx* switch.
5. If the CONSTANT option is used, *literal-1* must also be used. This option provides another means of defining constants that may be used anywhere in the program that a literal could be specified.
6. The remaining options of the >>SET CDF directive provide equivalent functionality to the >>DEFINE [how ever as this form is no longer in the Cobol standards and is classsed as archaic and wil be so flagged, the use of >>DEFINE should be used for all new programs], and >>SOURCE directives, as follows:

```
>>SET cdf-variable-1
          >>DEFINE cdf-variable-1 AS OFF

>>SET cdf-variable-1 AS literal-1
          >>DEFINE cdf-variable-1 AS literal-1

>>SET CONSTANT cdf-variable-1 literal-1
          >>DEFINE CONSTANT cdf-variable-1 literal-1

>>SET SOURCEFORMAT AS FIXED
          >>SOURCE FORMAT IS FIXED
```

```
>>SET SOURCEFORMAT AS FREE
          >>SOURCE FORMAT IS FREE

>>SET SOURCEFORMAT AS VARIABLE
          >>SOURCE FORMAT IS

>>SET SOURCEFORMAT AS XOPEN
          >>SOURCE FORMAT IS XOPEN

>>SET SOURCEFORMAT AS XCARD
          >>SOURCE FORMAT IS XCARD

>>SET SOURCEFORMAT AS CRT
          >>SOURCE FORMAT IS CRT

>>SET SOURCEFORMAT AS TERMINAL
          >>SOURCE FORMAT IS TERMINAL

>>SET SOURCEFORMAT AS COBOLX
          >>SOURCE FORMAT IS COBOLX

>>SET XFD literal-1
          [See chapter 13]

>>SET Micro-Focus-Directive
          [to do]
```

3.7. >>SOURCE

CDF >>SOURCE Directive Syntax

```
>>SOURCE FORMAT IS { FIXED|FREE|VARIABLE|XOPEN|XCARD|CRT|TERMINAL|COBOLX }
~~~~~~~~         ~~~~~ ~~~~ ~~~~~~~~ ~~~~~ ~~~~~ ~~~ ~~~~~~~~ ~~~~~~
```

The `>>SOURCE` CDF directive puts the compiler into `FIXED` or `FREE` source-code format mode. This, in effect, provides yet another mechanism for controlling the compiler's `-free` switch and `-fixed` switch.

1. The reserved words `FORMAT` and `IS` are optional and may be included, or not, at the discretion of the programmer. The presence or absence of these words has no effect upon the program.
2. FIXED : Source code is divided into: columns 1-6, the sequence number area; column 7, the indicator area; columns 8-72, the program-text area; and columns 72-80 as the reference area.
3. FREE : Source code text area starts in column 1 and continues till the end of line (to a maximum of 255 characters).
4. VARIABLE : Identical to FIXED format above except that it extends up to column 256 (in MF and some others, it is 252).
5. XOPEN : X/Open Free-form format. The program-text area may start in column 1 unless an indicator is present, and lines may contain up to 80 characters. Indicator for debugging lines is D instead of D or d.
6. XCARD : ICOBOL xCard format. Variable format with right margin set at column 255 instead of 250.
7. CRT : ICOBOL Free-form format (CRT). Similar to the X/Open format above, with lines containing up to 320 characters and single-character debugging line indicators (D or d).
8. TERMINAL : ACUCOBOL-GT Terminal format. Similar to the CRT format above, with indicator for debugging lines being \D instead of D or d. This format is mostly compatible with VAX COBOL terminal source format.
9. COBOLX : This format is similar to the CRT format above, except that the indicator area is always present in column 1; the program-text area starts in column 2 and extends up to the end of the record. Lines may contain up to 255 characters.
10. Note that with source formats XOPEN, CRT, TERMINAL, and COBOLX, missing spaces are not inserted within continued alphanumeric literals that are truncated before the right margin
11. You may switch between the various format modes as desired and it takes effect immediately.
12. You may also use the `>>SET` CDF directive to perform this function.
13. If the compiler is already in the specified mode, this statement will have no effect.

3.8. >>TURN

CDF >>TURN Directive Syntax

```
>>TURN { exception-name-1 [ file-name-1 ]...  }...
~~~~~~
   { OFF                            }
   { ~~~                            }
   { CHECKING ON [ WITH LOCATION ] }
     ~~~~~~~~ ~~         ~~~~~~~~
```

The directive will (de-)activate exception checks.

3.9. >>D

CDF >>D Directive Syntax

```
>>D
~~~
```

The directive removes all floating debug lines if debug mode not active. Otherwise will ignore the directive part of the line.

3.10. >>DISPLAY

CDF >>DISPLAY Directive Syntax

```
>>DISPLAY source-text [ VCS = version-string ]
~~~~~~~~~               ~~~
```

The directive is a v1.0 extension and will display messages during compilation.

3.11. >>PAGE

CDF >>PAGE Directive Syntax

```
>>PAGE [ comment-text ]
~~~~~~
```

The PAGE directive specifies page ejection and provides documentation for the source listing.

1. Comment-Text may contain any character in the computers standard character coded set except for control characters.
2. Comment-Text is not checked for syntax and only serves as documentation.
3. if a source listing is being produced a PAGE directive shall cause page ejection followed by listing of the PAGE directive.
4. If a listing is not being produced the directive will have no effect.

3.12. >>LISTING

CDF >>LISTING Directive Syntax

```
>>LISTING  {ON}
~~~~~~~~~  {OFF}
```

The directive allows the program listing to be de-(activated).

3.13. >>LEAP-SECONDS

CDF >>LEAP-SECONDS Directive Syntax

```
>>LEAP-SECONDS
  ~~~~~~~~~~~~~
```

The `>>LEAP-SECONDS` CDF directive is syntactically recognized but is otherwise non-functional.

Allows for more than 60 seconds per minute.

3.14. $ Directives

CDF $ Directive Syntax

```
$ (Dollar) Directives - Active.

These directives are active and have the same function as ones starting with >>:

$DEFINE
$DISPLAY ON|OFF
$IF
$ELIF
$ELSE
$ELSE-IF
$END
$SET

It is recommended to use the standard directives only instead of the MF
directives (when possible) as these have a higher chance for being portable.

$ (Dollar) Directives - Not Active.
These are NOT active and will produce a warning message:

$DISPLAY VCS ...

Recognised but otherwise ignored.

@OPTIONS options-text

Additional Micro-Focus directives accepted :

ADDRSV | ADD-RSV literal-1
ADDSYN | ADD-SYN literal-1 = literal-2
ASSIGN  "EXTERNAL" | "DYNAMIC"
BOUND
CALLFH  literal-1
COMP1  |  COMP-1  "BINARY" | "FLOAT"
FOLDCOPYNAME | FOLD-COPY-NAME  AS "UPPER" | "LOWER"
MAKESYN  |  MAKE-SYN
NOBOUND  |  NO-BOUND
NOFOLDCOPYNAME  |  NOFOLD-COPY-NAME  |  NO-FOLD-COPY-NAME
OVERRIDE  literal-1 = literal-2
REMOVE  literal-1
SOURCEFORMAT | SOURCE-FORMAT "FIXED" | "FREE" | "VARIABLE"
SSRANGE "2"
```

`NOSSRANGE | NO-SSRANGE`

Offers support for MF Compiler Directives.

3.15. Predefined compilation variables

GnuCOBOL defines *compilation variables* when various conditions are true. If the condition associated with a variable is false, the variable is not defined.

`DEBUG` The -d debug flag is specified.

`EXECUTABLE`
Module being compiled contains the main program.

`GCCOMP` The size of a `COMP` *item* is determined according to the GnuCOBOL scheme where for a picture of length:

1 - 2	*item* = 1 byte
3 - 4	*item* = 2 bytes
5 - 9	*item* = 4 bytes
10 - 18	*item* = 8 bytes

`GNUCOBOL` GnuCOBOL is compiling the source unit.

`HOSTSIGNS`
A signed packed decimal item's value may be considered `NUMERIC` if sign = `X"F"`.

`IBMCOMP` The size of a COMP *item* is determined according to the IBM scheme, where for a `PICTURE` of length:

1 - 4	*item* = 2 bytes
5 - 9	*item* = 4 bytes
10 - 18	*item* = 8 bytes

`MODULE` The element being compiled does not contain the main program.

`NOHOSTSIGNS`
A signed packed decimal item's value may *not* be considered `NUMERIC` if sign = `X"F"`.

`NOIBMCOMP`
The size of a `COMP` item is *not* determined according to the IBM scheme.

`NOSTICKY-LINKAGE`
Sticky linkage (linkage section items remaining allocated between invocations) is *not* active.

`NOTRUNC` Numeric data items are truncated according to their internal representation.

`P64` Pointers are greater than 32 bits.

`STICKY-LINKAGE`
Sticky linkage (linkage section items remaining allocated between invocations) is active.

`TRUNC` Numeric data items are truncated according to their `PICTURE` clauses.

While still supported, this may well be removed in the future and should not be used. See `GCCOMP` and `GNUCOBOL` instead:

`OCCOMP` The size of a `COMP` *item* is determined according to the GnuCOBOL scheme, where for a `PICTURE` of length:

1 - 2	*item* = 1 byte
3 - 4	*item* = 2 bytes
5 - 9	*item* = 4 bytes
10 - 18	*item* = 8 bytes

`OPENCOBOL`

GnuCOBOL is compiling the source unit.

End of Chapter 3 — CDF - Compiler Directing Facility

4. IDENTIFICATION DIVISION

IDENTIFICATION DIVISION Syntax

```
[{ IDENTIFICATION } DIVISION. ]
 { ~~~~~~~~~~~~~~ } ~~~~~~~~
 { ID             }
   ~~
 { PROGRAM-ID.  } { program name } .
 { ~~~~~~~~~~   } { literal-1    } [ AS { literal-2 } ] [ Type-clause ] .
 { FUNCTION-ID. } { literal-3 } [ AS literal-4 ] .
   ~~~~~~~~~~~    { function-name } .
 { OPTIONS. }
   ~~~~~~~
 [ DEFAULT ROUNDED MODE IS {AWAY-FROM-ZERO         }
   ~~~~~~~ ~~~~~~~         {NEAREST-AWAY-FROM-ZERO }
                           {NEAREST-EVEN           }
                           {NEAREST-TOWARDS-ZERO   }
                           {PROHIBITED             }
                           {TOWARDS-GREATER        }
                           {TOWARDS-LESSER         }
                           {TRUNCATION             }]
 [ ENTRY-CONVENTION IS {COBOL   }
   ~~~~~~~~~~~~~~~~    {EXTERN  }
                       {STDCALL }]
 [ AUTHOR.         comment-1.  ]
   ~~~~~~
 [ DATE-COMPILED.  comment-2.  ]
   ~~~~~~~~~~~~~
 [ DATE-MODIFIED.  comment-3.  ]
   ~~~~~~~~~~~~~
 [ DATE-WRITTEN.   comment-4.  ]
   ~~~~~~~~~~~~
 [ INSTALLATION.   comment-5.  ]
   ~~~~~~~~~~~~
 [ REMARKS.        comment-6.  ]
   ~~~~~~~
 [ SECURITY.       comment-7.  ]
   ~~~~~~~~
```

The AUTHOR, DATE-COMPILED, DATE-MODIFIED, DATE-WRITTEN, INSTALLATION, REMARKS and SECURITY paragraphs are supported by GnuCOBOL only to provide compatibility with programs written for the ANS1974 (or earlier) standards. As of the ANS1985 standard, these clauses have become obsolete and should not be used in new programs.

PROGRAM-ID Type Clause Syntax

```
IS [ COMMON ] [ INITIAL|RECURSIVE PROGRAM ]
     ~~~~~~     ~~~~~~~ ~~~~~~~~~
```

The identification division provides basic identification of the program by giving it a name and optionally defining some high-level characteristics via the eight pre-defined paragraphs that may be specified.

1. The paragraphs shown above may be coded in any sequence.
2. The reserved words `AS`, `IS` and `PROGRAM` are optional and may be included, or not, at the discretion of the programmer. The presence or absence of these words has no effect upon the program.
3. A **Type Clause** may be coded only when `PROGRAM-ID` is specified. If one is coded, either `COMMON`, `COMMON INITIAL` or `COMMON RECURSIVE` must be specified.
4. While the actual `IDENTIFICATION DIVISION` or `ID DIVISION` header is optional, the `PROGRAM-ID` / `FUNCTION-ID` paragraphs are not; only one or the other, however, may be coded.
5. The compiler's `-Wobsolete` switch will cause the GnuCOBOL compiler to issue warnings messages if these (or any other obsolete syntax) is used in a program.
6. If specified, *literal-1* must be an actual alphanumeric literal and may not be a figurative constant.
7. The `PROGRAM-ID` and `FUNCTION-ID` paragraphs serve to identify the program to the external (i.e. operating system) environment. If there is no `AS` clause present, the *program-id* will serve as that external identification. If there is an `AS` clause specified, that specified literal will serve as the external identification. For the remainder of this document, that "external identification" will be referred to as the primary entry-point name.
8. The `INITIAL`, `COMMON` and `RECURSIVE` words are used only within subprograms serving as subroutines. Their purposes are as follows:
   A. `COMMON` should be used only within subprograms that are nested subprograms. A nested subprogram declared as `COMMON` may be called from any nested program in the source file being compiled, not just those "above" it in the nesting structure.
   B. The `RECURSIVE` clause, if any, will cause the compiler to generate different object code for the subprogram that will enable it to invoke itself and to properly return back to the program that invoked it.

      User-defined functions (i.e. `FUNCTION-ID`) are always recursive.
   C. The `INITIAL` clause, if specified, guarantees the subprogram will be in its initial (i.e. compiled) state each and every time it is executed, not just the first time.

End of Chapter 4 — IDENTIFICATION DIVISION

5. ENVIRONMENT DIVISION

ENVIRONMENT DIVISION Syntax

```
  ENVIRONMENT DIVISION.
  ~~~~~~~~~~~ ~~~~~~~~
[ CONFIGURATION SECTION. ]
  ~~~~~~~~~~~~~ ~~~~~~~~
[ SOURCE-COMPUTER.         Compilation-Computer-Specification .  ]
  ~~~~~~~~~~~~~~~
[ OBJECT-COMPUTER.         Execution-Computer-Specification .  ]
  ~~~~~~~~~~~~~~~
[ SPECIAL-NAMES.           Program-Configuration-Specification .  ]
  ~~~~~~~~~~~~~
[ REPOSITORY.              Function-Specification...  .  ]
  ~~~~~~~~~~
[ INPUT-OUTPUT SECTION. ]
  ~~~~~~~~~~~~ ~~~~~~~
[ FILE-CONTROL.            General-File-Description...  .  ]
  ~~~~~~~~~~~~
[ I-O-CONTROL.             File-Buffering Specification...  .  ]
  ~~~~~~~~~~~
```

This division defines the external computer environment in which the program will be operating. This includes defining any files that the program may be .

- If both optional sections of this division are coded, they must be coded in the sequence shown.
- The paragraphs within the sections may be coded in any order.
- These sections consist of a series of specific, pre-defined, paragraphs (`SOURCE-COMPUTER` and `OBJECT-COMPUTER`, for example), each of which serves a specific purpose. If no code is required for the purpose one of the paragraphs serves, the entire paragraph may be omitted.
- If any of the paragraphs within one of the sections are coded, the section header itself must be coded.
- If none of the paragraphs within one of the sections are coded, the section header itself may be omitted.
- If none of the sections within the environment division are coded, the `ENVIRONMENT DIVISION.` header itself may be omitted.

5.1. CONFIGURATION SECTION

CONFIGURATION SECTION Syntax

```
  CONFIGURATION SECTION.
  ~~~~~~~~~~~~~ ~~~~~~~
[ SOURCE-COMPUTER. Compilation-Computer-Specification .  ]
  ~~~~~~~~~~~~~~~
[ OBJECT-COMPUTER. Execution-Computer-Specification .  ]
  ~~~~~~~~~~~~~~~
[ SPECIAL-NAMES.   Program-Configuration-Specification .  ]
  ~~~~~~~~~~~~~
[ REPOSITORY.      Function-Specification...  .  ]
  ~~~~~~~~~~
```

This section defines the computer system upon which the program is being compiled and executed and also specifies any special environmental configuration or compatibility characteristics.

1. The four paragraphs in this section may be specified in any order but if not in this order, a warning will be issued.
2. The configuration section is not allowed in a nested subprogram. A nested program inherits the configuration section settings of its parent program.
3. If none of the features provided by the configuration section are required by a program, the entire `CONFIGURATION SECTION.` header may be omitted from the program.

5.1.1. SOURCE-COMPUTER

SOURCE-COMPUTER Syntax

```
SOURCE-COMPUTER. computer-name [ WITH DEBUGGING MODE ] .
~~~~~~~~~~~~~~~                  ~~~~~~~~~ ~~~~
```

This paragraph defines the computer upon which the program is being compiled and provides one way in which debugging code embedded within the program may be activated.

1. The reserved word `WITH` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. This paragraph is not allowed in a nested subprogram. A nested program inherits the `SOURCE-COMPUTER` settings of its parent program.
3. The value specified for *computer-name* is irrelevant, provided it is a valid COBOL word that does not match any GnuCOBOL reserved word. The *computer-name* value may include spaces. This need not match the *computer-name* used with the `OBJECT-COMPUTER` paragraph, if any.
4. The `DEBUGGING MODE` clause, if present, will inform the compiler that debugging lines (those with a '`D`' in column 7 if Fixed Source Mode is in effect, or those prefixed with a `>>D` if Free Source Mode is in effect) — normally treated as comments — are to be compiled.
5. Even without the `DEBUGGING MODE` clause, it is still possible to compile debugging lines. Debugging lines may also be compiled by specifying the `-fdebugging-line` switch to the GnuCOBOL compiler.

5.1.2. OBJECT-COMPUTER

OBJECT-COMPUTER Syntax

```
OBJECT-COMPUTER.  [ computer-name ]
~~~~~~~~~~~~~~~

[ MEMORY SIZE IS integer-1 WORDS|CHARACTERS ]
  ~~~~~~ ~~~~              ~~~~~ ~~~~~~~~~~

[ PROGRAM COLLATING SEQUENCE IS alphabet-name-1 ]
          ~~~~~~~~

[ SEGMENT-LIMIT IS integer-2 ]
  ~~~~~~~~~~~~

[ CHARACTER CLASSIFICATION IS { locale-name-1  } ]
            ~~~~~~~~~~~~~~    { LOCALE         }
                              { ~~~~~~         }
                              { USER-DEFAULT   }
                              { ~~~~~~~~~~~~   }
                              { SYSTEM-DEFAULT }
                                ~~~~~~~~~~~~~~
```

.

The `MEMORY SIZE` and `SEGMENT-LIMIT` clauses are syntactically recognized but are otherwise non-functional.

This paragraph describes the computer upon which the program will execute.

1. The *computer-name*, if specified, must immediately follow the `OBJECT-COMPUTER` paragraph name. The remaining clauses may be coded in any sequence.
2. The reserved words `CHARACTER`, `IS`, `PROGRAM` and `SEQUENCE` are optional and may be omitted. The presence or absence of these words has no effect on the program.
3. The value specified for *computer-name*, if any, is irrelevant provided it is a valid COBOL word that does not match any GnuCOBOL reserved word. The *computer-name* may include spaces. This need not match the *computer-name* used with the `SOURCE-COMPUTER` paragraph, if any.
4. The `OBJECT-COMPUTER` paragraph is not allowed in a nested subprogram. A nested program inherits the `OBJECT-COMPUTER` settings of its parent program.
5. The `COLLATING SEQUENCE` clause allows you to specify a customized character collating sequence to be used when alphanumeric values are compared to one another. Data will still be stored in the character set native to the computer, but the logical sequence in which characters are ordered for comparison purposes can be altered from that defined by the computer's native character set. The *alphabet-name-1* you specify needs to be defined in the `SPECIAL-NAMES` (see [SPECIAL-NAMES], page 92) paragraph.
6. If no `COLLATING SEQUENCE` clause is specified, the collating sequence implied by the character set native to the computer (usually ASCII) will be used.

7. The optional `CLASSIFICATION` clause may be used to specify a locale for the environment in which the program will execute, for the purpose of influencing the upper-case and lower-case mappings of characters for the `UPPER-CASE` (see [UPPER-CASE], page 529) and `LOWER-CASE` (see [LOWER-CASE], page 477) intrinsic functions and the classification of characters for the `ALPHABETIC`, `ALPHABETIC-LOWER` and `ALPHABETIC-UPPER` class tests. The definitions of these classes is taken from the cultural convention specification (`LC_CTYPE`) from the specified locale.

   The meanings of the four locale specifications are as follows:

   A. *locale-name-1* references a `LOCALE` (see [SPECIAL-NAMES], page 92) definition.
   B. The keyword `LOCALE` refers to the current locale (in effect at the time the program is executed)
   C. The keyword `USER-DEFAULT` references the default locale specified for the user currently executing this program.
   D. The keyword `SYSTEM-DEFAULT` denotes the default locale specified for the computer upon which the program is executing.

8. Absence of a `CLASSIFICATION` clause will cause character classification to occur according to the rules for the computer's native character set (ASCII, EBCDIC, etc.).

5.1.3. SPECIAL-NAMES

SPECIAL-NAMES Syntax

```
SPECIAL-NAMES.
~~~~~~~~~~~~~
 [ CALL-CONVENTION integer-1 IS mnemonic-name-1 ]
   ~~~~~~~~~~~~~~~
 [ CONSOLE IS CRT ]
   ~~~~~~~    ~~~
 [ CRT STATUS IS identifier-1 ]
   ~~~ ~~~~~~
 [ CURRENCY SIGN IS literal-1 ]
   ~~~~~~~~ ~~~~
 [ CURSOR IS identifier-2 ]
   ~~~~~~
 [ DECIMAL-POINT IS COMMA ]
   ~~~~~~~~~~~~    ~~~~~
 [ EVENT STATUS IS identifier-3 ]
   ~~~~~ ~~~~~~
 [ LOCALE locale-name-1 IS literal-2 ]...
   ~~~~~~
 [ NUMERIC SIGN IS TRAILING SEPARATE ]
   ~~~~~~~ ~~~~    ~~~~~~~~ ~~~~~~~~
 [ SCREEN CONTROL IS identifier-4 ]
   ~~~~~~ ~~~~~~~
 [ device-name-1 IS mnemonic-name-2 ]...

 [ feature-name-1 IS mnemonic-name-3 ]...

 [ Alphabet-Clause ]...

 [ Class-Definition-Clause ]...

 [ Switch-Definition-Clause ]...

 [ Symbolic-Characters-Clause ]...
 .
```

The `EVENT STATUS` and `SCREEN CONTROL` clauses are syntactically recognized but are otherwise non-functional.

Alphabet-Name-Clause, *Class-Definition-Clause*,
Switch-Definition-Clause and *Symbolic-Characters-Clause*
are discussed in detail in the next four sections.

The SPECIAL-NAMES paragraph provides a means for specifying various program and operating environment configuration options.

1. The various clauses that may be specified within the SPECIAL-NAMES paragraph may be coded in any order.
2. The reserved word IS is optional and may be omitted. The presence or absence of this word has no effect upon the program.
3. The SPECIAL-NAMES paragraph is not allowed in a nested subprogram. A nested program inherits the SPECIAL-NAMES settings of its parent program.
4. Only the final clause specified within this paragraph should be terminated with a period.
5. The CALL-CONVENTION clause allows a decimal integer, representing a series of ON/OFF switch settings, to be associated with a mnemonic name which may then be coded on a CALL statement (see [CALL], page 293). The switch settings defined by this mnemonic will then control how the linkage to a subroutine invoked by the CALL statement that references *mnemonic-name-1* will be handled.
6. The CONSOLE IS CRT clause, if specified, will cause a DISPLAY statement lacking an explicit UPON clause to be treated as a DISPLAY data-item statement (see [DISPLAY data-item], page 311), and any ACCEPT statement lacking a FROM clause to be treated as a ACCEPT data-item statement (see [ACCEPT data-item], page 272).
7. If the CRT STATUS clause is not specified, an implicit COB-CRT-STATUS identifier (with a PICTURE 9(4)) will be allocated for the purpose of receiving screen ACCEPT statuses. If CRT STATUS is specified, then *identifier-1* must be defined in the program as a PICTURE 9(4) field.
8. The CURRENCY SIGN clause may be used to redefine the character to be used as a currency sign in a PICTURE (see [PICTURE], page 194) clause. The default currency sign is a dollar-sign ('$'). You may specify any character *except* 0-9, A-Z, a-z, +, -, ,, ., *, /, ;, (,), =, \\, quote ('"') or space.
9. The CURSOR IS clause allows you to specify a 4- or 6-character data item into which the cursor screen location at the time a screen ACCEPT is satisfied. The value will be returned as *rrcc* or *rrrccc*, depending upon the length of the specified *identifier-2*, where *rr* and *rrr* represent the row number (starting at zero) and *cc* and *ccc* represent the column number (also starting at zero). There is no default data item allocated for this data if the CURSOR IS clause is not specified, and it is the programmer's responsibility to define *identifier-2* if the clause is specified.
10. The DECIMAL POINT IS COMMA clause reverses the definition of the ',' and '.' characters when they are used as PICTURE editing symbols and within numeric literals. This can have unwanted side-effects - see [Punctuation], page 37.
11. The LOCALE clause may be used to associate external OS-defined locale names (*literal-2*) with an internal name (*locale-name-1*) that may then be referenced within the program. Locale names are defined by the Operating System and/or C compiler GnuCOBOL will be utilizing on your computer.
12. The following is the list of possible locale codes, for example, that would be available on a Windows computer running a GnuCOBOL version that was built utilizing the MinGW Unix-emulator and the GNU C compiler (gcc):

A	af_ZA, am_ET, ar_AE, ar_BH, ar_DZ, ar_EG, ar_IQ, ar_JO, ar_KW, ar_LB, ar_LY, ar_MA, ar_OM, ar_QA, ar_SA, ar_SY, ar_TN, ar_YE, arn_CL, as_IN, az_Cyrl_AZ, az_Latn_AZ
B	ba_R, be_BY, bg_BG, bn_IN bo_BT, bo_CN, br_FR, bs_Cyrl_BA, bs_Latn_BA
C	ca_ES, cs_CZ, cy_GB
D	da_DK, de_AT, de_CH, de_DE, de_LI, de_LU, dsb_DE, dv_MV
E	el_GR, en_029, en_AU, en_BZ, en_CA, en_GB, en_IE, en_IN, en_JM, en_MY en_NZ, en_PH, en_SG, en_TT, en_US, en_ZA, en_ZW, es_AR, es_BO, es_CL, es_CO, es_CR, es_DO, es_EC, es_ES, es_GT, es_HN, es_MX, es_NI, es_PA, es_PE, es_PR, es_PY, es_SV, es_US, es_UY es_VE, et_EE, eu_ES
F	fa_IR, fi_FI, fil_PH, fo_FO, fr_BE, fr_CA, fr_CH, fr_FR, fr_LU, fr_MC, fy_NL
G	ga_IE, gbz_AF, gl_ES, gsw_FR, gu_IN
H	ha_Latn_NG, he_IL, hi_IN, hr_BA, hr_HR, hu_HU, hy_AM
I	id_ID, ig_NG, ii_CN, is_IS, it_CH, it_IT, iu_Cans_CA, iu_Latn_CA
J	ja_JP
K	ka_GE, kh_KH, kk_KZ, kl_GL, kn_IN, ko_KR, kok_IN, ky_KG
L	lb_LU, lo_LA, lt_LT, lv_LV
M	mi_NZ, mk_MK, ml_IN, mn_Cyrl_MN, mn_Mong_CN moh_CA, mr_IN, ms_BN, ms_MY, mt_MT
N	nb_NO, ne_NP, nl_BE, nl_NL, nn_NO, ns_ZA
O	oc_FR, or_IN
P	pa_IN, pl_PL, ps_AF, pt_BR, pt_PT
Q	qut_GT, quz_BO, quz_EC, quz_PE
R	rm_CH, ro_RO, ru_RU, rw_RW
S	sa_IN, sah_RU, se_FI, se_NO se_SE, si_LK, sk_SK, sl_SI, sma_NO, sma_SE, smj_NO, smj_SE, smn_FI, sms_FI, sq_AL, sr_Cyrl_BA, sr_Cyrl_CS, sr_Latn_BA, sr_Latn_CS, sv_FI, sv_SE, sw_KE syr_SY
T	ta_IN, te_IN, tg_Cyrl_TJ, th_TH tk_TM, tmz_Latn_DZ, tn_ZA, tr_IN, tr_TR, tt_RU
U	ug_CN, uk_UA, ur_PK, uz_Cyrl_UZ, uz_Latn_UZ
V	vi_VN
W	wen_DE, wo_SN
X	xh_ZA

Y yo_NG

Z zh_CN, zh_HK, zh_MO, zh_SG, zh_TW, zu_ZA

13. The `NUMERIC SIGN TRAILING SEPARATE` specification causes all signed numeric `USAGE DISPLAY` data items to be created as if the `SIGN IS TRAILING SEPARATE CHARACTER` clause was included in their definitions.

14. The *`device-name-1`* `IS` *`mnemonic-name-2`* clause allows you to specify an alternate name (*device-name-1*) for one of the built-in GnuCOBOL device name *mnemonic-name-2*. The list of device names built-into GnuCOBOL, and the physical device associated with that name, are as follows:

    `CONSOLE` This is the (screen-mode) display of the PC or Unix system.

    `STDIN`
    `SYSIN`
    `SYSIPT` These devices (they are all synonymous) represent standard system input (pipe 0). On a PC or UNIX system, this is typically the keyboard. The contents of a file may be delivered to a GnuCOBOL program for access via one of these device names by adding the sequence '`0< filename`' to the end of the programs execution command.

    `PRINTER`
    `STDOUT`
    `SYSLIST`
    `SYSLST`
    `SYSOUT` These devices (they are all synonymous) represent standard system output (pipe 1). On a PC or UNIX system, this is typically the display. Output sent to one of these devices by a GnuCOBOL program can be sent to a file by adding the sequence '`1> filename`' to the end of the programs execution command.

    `STDERR`
    `SYSERR` These devices (they are synonymous) represent standard system error output (pipe 2). On a PC or UNIX system, this is typically the display. Output sent to one of these devices by a GnuCOBOL program can be sent to a file by adding the sequence '`2> filename`' to the end of the programs execution command.

15. The *`feature-name-1`* `IS` *`mnemonic-name-3`* clause allow for mnemonic names to be assigned to up to the 13 printer channel (i.e. vertical page positioning) position feature names `C`*`nn`* (*`nn`*`=01-12`) and `CSP`. Once a channel position has been assigned a mnemonic name, statements of the form `WRITE` *`record-name`* `AFTER ADVANCING` *`mnemonic-name-3`* may be coded to write the specified print record at the channel position assigned to *mnemonic-name-3*.

    Printers supporting channel positioning are generally mainframe-type line printers. When writing to printers that do not support channel positioning, a formfeed will be issued to the printer.

    The `CSP` positioning option stands for "No Spacing". Testing on a MinGW build of GnuCOBOL shows that this too results in a formfeed being issued.

5.1.3.1. Alphabet-Name-Clause

SPECIAL-NAMES Alphabet-Clause Syntax

```
ALPHABET alphabet-name-1 IS { ASCII              }
~~~~~~~~                    { ~~~~~              }
                            { EBCDIC             }
                            { ~~~~~~             }
                            { NATIVE             }
                            { ~~~~~~             }
                            { STANDARD-1         }
                            { ~~~~~~~~~~         }
                            { STANDARD-2         }
                            { ~~~~~~~~~~         }
                            { Literal-Clause...  }
```

SPECIAL-NAMES ALPHABET Literal-Clause Syntax

```
literal-1 [ { THRU|THROUGH literal-2 } ]
            { ~~~~ ~~~~~~~           }
            { {ALSO literal-3}...    }
               ~~~~
```

The `ALPHABET` clause relates *alphabet-name-1* to a specified character code set or collating sequence, including one you define yourself using the *literal-1* option.

1. The reserved word `IS` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. The reserved words `THRU` and `THROUGH` are interchangeable.
3. GnuCOBOL considers `ASCII`, `STANDARD-1` and `STANDARD-2` to be interchangeable.
4. `NATIVE` specifies the system default character set.
5. The following points apply to using the *literal-n* specifications to compose a custom character set:
   A. The *literal-n* values are either integers or alphanumeric quoted characters. These represent a single character in the `NATIVE` character set, either by its actual text value (alphanumeric quoted character) or by ordinal position in the `NATIVE` character set (integer),
   B. The sequence in which characters are defined in this clause specifies the relative order those characters should have when comparisons are made using this alphabet.
   C. Character positions in this list do not affect the actual binary storage values used for the characters. Binary values will still be those of the `NATIVE` character set.

   D. You may specify any of the figurative constants `SPACE`, `SPACES`, `ZERO`, `ZEROS`, `ZEROES`, `QUOTE`, `QUOTES`, `HIGH-VALUE`, `HIGH-VALUES`, `LOW-VALUE` or `LOW-VALUES` for any of the *literal-1*, *literal-2* or *literal-3* specifications.

6. Once you have defined an alphabet name, that alphabet name may be used on specifications in `CODE-SET`, `COLLATING SEQUENCE`, or `SYMBOLIC CHARACTERS` clauses elsewhere in the program.

5.1.3.2. Class-Definition-Clause

SPECIAL-NAMES Class-Definition-Clause Syntax

```
CLASS class-name-1 IS { literal-1 [ THRU|THROUGH literal-2 ] }...
~~~~~                              ~~~~ ~~~~~~~
```

---

1. The reserved word `IS` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. The reserved words `THRU` and `THROUGH` are interchangeable.
3. Both *literal-1* and *literal-2* must be alphanumeric literals of length 1.
4. The literal(s) specified on this clause define the possible characters that may be found in a data item's value in order to be considered part of the class.
5. For example, the following defines a class called `Hexadecimal`, the definition of which specifies the only characters that may be present in an alphanumeric data item if that data item is to be part of the `Hexadecimal` class:

```
CLASS Hexadecimal IS '0' THRU '9'
                     'A' THRU 'F'
                     'a' THRU 'f'
```

6. Once class `Hexadecimal` has been defined, program code could then use a statement such as `IF input-item IS Hexadecimal` to determine if the value of characters in a data item are valid according to that class.

5.1.3.3. Switch-Definition-Clause

SPECIAL-NAMES Switch-Definition-Clause Syntax

```
switch-name-1 [ IS mnemonic-name-1 ]

  [ ON STATUS IS condition-name-1 ]
    ~~
  [ OFF STATUS IS condition-name-2 ]
    ~~~
```

The switch-definition clause associates a condition-name with a run-time execution switch so that the status of that switch may be tested from within a program.

1. The reserved words `IS` and `STATUS` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The valid *switch-name-1* names are `SWITCH-n` (*n* = 0-36).
3. If the program is compiled with the `-fsyntax-extension` switch, the switch names `SWn` (*n* = 0-15) are also valid; they correspond to `SWITCH-0` through `SWITCH-15`, respectively as well as `SWITCH-16` through `SWITCH-36`, `SWITCH 0` through `SWITCH 26` and `SWITCH A` through `SWITCH Z`.
4. At execution time, each switch will be associated with a `COB_SWITCH_n` run-time environment variable, where *n* will have the value '`0`' through '`15`'. Any of these sixteen environment variables that have the value `ON` (regardless of upper- or lower-case value) will be considered to be set "on". Any of these sixteen environment variables having no value at all or a value other than `ON` will be considered `OFF`.
5. Each specified switch must have at least one of a `IS` *mnemonic-name-1*, `ON STATUS` or an `OFF STATUS` option defined for it, otherwise there will be no way to reference the switch from within a GnuCOBOL program.
6. The `IS` *mnemonic-name-1* syntax provides a means for setting the switch to either an `ON` or `OFF` value via the `SET` statement (see [SET], page 382).
7. The `ON STATUS` and `OFF STATUS` syntax provides a way of associating a condition-name with either the on or off status of the switch, so that status may be tested at execution time via the `IF` statement (see [IF], page 338).

5.1.3.4. Symbolic-Characters-Clause

SPECIAL-NAMES-Symbolic-Characters-Clause Syntax

```
SYMBOLIC CHARACTERS
~~~~~~~~
  { symbolic-character-1...  IS|ARE integer-1...  }...

  [ IN alphabet-name-1 ]
    ~~
```

This clause may be used to define your own figurative constants.

1. The reserved words `ARE`, `CHARACTERS` and `IS` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. There must be exactly as many *integer-1* values specified as there are *symbolic-character-1* names.
3. Each symbolic character name will be associated with the corresponding *integer-1*th character in the alphabet named in the `IN` clause. The integer values are selecting characters from the alphabet by their ordinal position and not by their numeric value; thus, an integer of 15 will select the 15th character in the specified alphabet, regardless of the actual numeric value of the bit pattern that constitutes that character.
4. If no *alphabet-name-1* is specified, the systems native character set will be assumed.
5. The following two code examples define the same set of figurative constant names for five ASCII control characters (assuming that ASCII is the system's native character set). The two examples are identical in their effects, even though the way the figurative constants are defined is different.

   Individually

   ```
   SYMBOLIC CHARACTERS NUL IS 1
                       SOH IS 2
                       BEL IS 8
                       DC1 IS 18
                       DC2 IS 19
   ```

   Respectively

   ```
   SYMBOLIC CHARACTERS NUL SOH BEL DC1 DC2
                   ARE   1   2   8  18  19
   ```

5.1.4. REPOSITORY

REPOSITORY Syntax

```
REPOSITORY.
~~~~~~~~~~
   FUNCTION { function-prototype-name-1 [ AS literal-1 ] }...
   ~~~~~~~~ {                             ~~            }
            { intrinsic-function-name-1 [ AS literal-2 ] }
            {                             ~~            }
            { intrinsic-function-name-2 INTRINSIC       }
            { ALL INTRINSIC             ~~~~~~~~~       }
              ~~~ ~~~~~~~~~
```

The REPOSITORY paragraph provides a way to control access to the various built-in intrinsic functions and any user defined functions that your program will be using.

1. The `REPOSITORY` paragraph is not allowed in a nested subprogram. A nested program inherits the `REPOSITORY` settings of its parent program.
2. The `INTRINSIC` clause allows you to flag one or more (or `ALL`) built-in intrinsic functions as being usable without the need to code the keyword `FUNCTION` in front of the function names.
3. As an alternative to using the `ALL INTRINSIC` clause, you may instead compile your GnuCOBOL programs using the `-fintrinsics=ALL` switch.
4. The *function-prototype-name-1* option is required to specify the name of a user-defined function your program will be using. Optionally, should you desire, you may specify an alias name by which you will reference that user-defined function. Should you wish, you may also use the `AS` clause to provide an alias name for a built-in intrinsic function.
5. The following example
   - enables all intrinsic functions to be specified without the use of the `FUNCTION` keyword,
   - names two user-defined functions named `MY-FUNCTION-1` and `MY-FUNCTION-2` that will be used by the program and
   - specifies the alias names `SIGMA` for the intrinsic function `STANDARD-DEVIATION` and `MF2` for `MY-FUNCTION-2`.

   ```
   REPOSITORY.
       FUNCTION ALL INTRINSIC
       FUNCTION MY-FUNCTION-1
       FUNCTION MY-FUNCTION-2 AS "MF2"
       FUNCTION STANDARD-DEVIATION AS "SIGMA".
   ```

A special note about user-defined functions — because you must name a user-defined function that your program will be using in the `REPOSITORY` paragraph, you may always

reference that function from your program's procedure division without needing to use the `FUNCTION` keyword.

5.2. INPUT-OUTPUT SECTION

INPUT-OUTPUT SECTION Syntax

```
[ INPUT-OUTPUT SECTION. ]
  ~~~~~~~~~~~~ ~~~~~~~
[ FILE-CONTROL. ]
  ~~~~~~~~~~~~
    [ SELECT-Statement...  ]

[ I-O-CONTROL. ]
  ~~~~~~~~~~~
    [ MULTIPLE-FILE-Statement ]

    [ SAME-RECORD-Statement ]
```

The `INPUT-OUTPUT` section provides for the definition of any files the program will be accessing as well as control of the I/O buffering process against those files through the `FILE-CONTROL` and `I-O-CONTROL` paragraphs, respectively.

1. As the diagram shows, there are three types of statements that may occur in the two paragraphs of this section. If none of the statements are coded in a particular paragraph, the paragraph itself may be omitted, otherwise it is required.
2. If neither paragraph is coded, the `INPUT-OUTPUT SECTION.` header itself may be omitted, otherwise it is normally required.
3. If the compiler *configuration file* (see [Compiler Configuration Files], page 649) you are using has `relaxed-syntax-check` set to '`yes`', the `FILE-CONTROL` and `I-O-CONTROL` paragraphs may be specified without the `INPUT-OUTPUT SECTION` header having been coded.
4. If both statement types are coded in the `I-O-CONTROL` paragraph, the order in which those statements are coded is irrelevant.

5.2.1. SELECT

SELECT Statement Syntax

```
SELECT [ [ NOT ] OPTIONAL ] file-name-1
~~~~~~     ~~~   ~~~~~~~~
[ ASSIGN { TO    } [{ EXTERNAL }] [{ DISC|DISK      }] [{ identifier-1 }] ]
  ~~~~~~ { USING }  { ~~~~~~~~ }   { ~~~~ ~~~~      }   { word-1       }
                    { DYNAMIC  }   { DISPLAY        }   { literal-1    }
                      ~~~~~~~      { ~~~~~~~        }
                                   { KEYBOARD       }
                                   { ~~~~~~~~       }
                                   { LINE ADVANCING }
                                   { ~~~~ ~~~~~~~~~ }
                                   { PRINTER        }
                                   { ~~~~~~~        }
                                   { RANDOM         }
                                   { ~~~~~~         }
                                   { TAPE           }
                                     ~~~~
[ COLLATING SEQUENCE IS alphabet-name-1 ]
  ~~~~~~~~~
[ FILE|SORT ] STATUS IS identifier-2 [ identifier-3 ] ]
  ~~~~ ~~~~   ~~~~~~
[ LOCK MODE IS { MANUAL|AUTOMATIC                                  } ]
  ~~~~         { ~~~~~~ ~~~~~~~~~                                  }
               { EXCLUSIVE [ WITH { LOCK ON MULTIPLE RECORDS } ] }
                 ~~~~~~~~~        { ~~~~ ~~ ~~~~~~~~ ~~~~~~~ }
                                  { LOCK ON RECORD            }
                                  { ~~~~ ~~ ~~~~~~            }
                                  { ROLLBACK                  }
                                  { ~~~~~~~~                  }
[ ORGANIZATION Clause ]
  ~~~~~~~~~~~~
[ ORGANISATION Clause ]
  ~~~~~~~~~~~~
[ RECORD DELIMITER IS STANDARD-1 ]
  ~~~~~~ ~~~~~~~~~    ~~~~~~~~~~
[ RESERVE integer-1 AREAS ]
  ~~~~~~~
[ SHARING WITH { ALL OTHER } ]
  ~~~~~~~      { ~~~       }
               { NO OTHER  }
               { ~~        }
               { READ ONLY }
```

~~~~ ~~~~

The COLLATING SEQUENCE, RECORD DELIMITER, RESERVE and ALL OTHER clauses are syntactically recognized but are otherwise non-functional.

The SELECT statement creates a definition of a file and links that COBOL definition to the external operating system environment.

1. The reserved words AREAS, IS, MODE, OTHER, SEQUENCE, TO, USING and WITH are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. After *file-name-1*, the various clauses may be coded in any sequence.
3. A period must follow the last coded clause.
4. The OPTIONAL clause, to be used only for files that will be used to provide input data to the program, indicates the file may or may not actually be available at runtime. Attempts to OPEN an OPTIONAL file when the file does not exist will receive a special non-fatal file status value (see status 05 in the list of file status values below) indicating the file is not available; a subsequent attempt to READ that file will return an AT END (end-of-file) condition. Optionally, files may be designated as NOT OPTIONAL, if desired. This is useful when specifying the compiler's -foptional-file switch, which automatically makes all files OPTIONAL except for those explicitly declared as NOT OPTIONAL.
5. The *file-name-1* value that you specify will be the name by which you will reference the file within your program. This name should be formed according to the rules for user-defined names (see [User-Defined Words], page 10).
6. The optional ASSIGN clause specifies how — at runtime, when *file-name-1* is opened — either a logical device (STDIN, STDOUT) or a file anywhere in one of the currently-mounted file systems will be associated with *file-name-1*, as follows:
   A. There are three components to the ASSIGN clause:

Type	EXTERNAL, DYNAMIC or neither
Device	the list of device choices
Locator	shown as a choice between *identifier-1*, *word-1* and *literal-1*.

   B. ASSIGN TO DISC *file-name-1* will be assumed if there is no ASSIGN clause on a SELECT.
   C. If an ASSIGN clause is coded without a *Device*, the device DISC will be assumed.
   D. If a *Locator* clause is coded, the COBOL file *file-name-1* will be attached to a data file within any file system that is mounted and available to the executing program at the time *file-name-1* is opened. How that file is identified varies, depending upon the specified *Locator*, as follows:
      a. If *literal-1* is coded, the value of the literal will serve as the File Location String that will identify the data file.
      b. If *identifier-1* is coded, the value of the identifier will serve as the File Location String that will identify the data file.
~~~~

c. If *word-1* (a syntactically valid word not duplicating a reserved or user-defined word) is coded, and a *Type* is `EXTERNAL`, then *word-1* itself will serve as the File Location String that will identify the data file. If, however, a *Type* of `EXTERNAL` was *not* specified, the compiler will create a `PIC X(1024)` data item named *word-1* within the program; the contents of that data item at the time the program opens *file-name-1* will then serve as the File Location String that will identify the data file.

d. File Location Strings will be discussed shortly.

E. If no *Locator* is coded, *file-name-1* will be attached to a logical device or a file based upon the specified (or implied) *Device*, as follows:

a. `DISC` or `DISK` will assume an attachment to a file named *file-name-1* in whatever directory is current at the time the file is opened.

b. `DISPLAY` will assume an attachment to the `STDOUT` logical device; these files should only be used for output.

c. `KEYBOARD` will assume an attachment to the `STDIN` logical device; these files should only be used for input.

d. `PRINTER` will assume an attachment to the `LPT1` logical device/port; these files should only be used for output.

e. `RANDOM` or `TAPE` will behave exactly as `DISC` does. These two additional *Device*s are provided to facilitate the compilation of COBOL source from other COBOL implementations.

F. The `LINE ADVANCING` device *requires* that a *Locator* be specified; these files should only be used for output. A COBOL Line Advancing file will allow carriage-control characters such as line-feeds and form-feeds to be written to the attached operating system file, via the `ADVANCING` clause of the `WRITE` statement (see [WRITE], page 417).

G. File Location Strings are used (at runtime) to identify the path and filename to the data file that must be attached to *file-name-1* when that file is opened.

H. If the compiler *configuration file* (see [Compiler Configuration Files], page 649) you used to compile the program with had a `filename-mapping` value of `yes`, the GnuCOBOL runtime system will first attempt to identify a currently-defined environment variable whose value will serve as the data file's path and filename, as follows:

a. If the compiler *configuration file* (see [Compiler Configuration Files], page 649) (see [Compiler Configuration Files], page 649) you used to compile the program specified `mf` as the `assign-clause` value, then the File Locator String will be interpreted according to Microfocus COBOL rules — namely, everything before the last '`-`' in the File Locator String will be ignored; the characters after the last '`-`' will be treated as the base of an environment variable name. If there is no '`-`' character in the File Locator String then the entire File Locator String will serve as the base of an environment variable name. This is the default behaviour for every config file except `ibm`.

b. If, on the other hand, the compiler *configuration file* (see [Compiler Configuration Files], page 649) you used to compile the program specified `mf` as the

`assign-clause` value, then the File Locator String will be interpreted according to according to IBM COBOL rules — namely, the File Locator String is expected to be of the form `S-xxx` or `AS-xxx`, in which case the *xxx* will be treated as the base of an environment variable name. If there is no '`-`' character in the File Locator String then the entire File Locator String will serve as the base of an environment variable name.

c. Once an environment variable name base (let's refer to it as *bbbb*) has been determined, the runtime system will look for the first one of the following environment variables that exists, in this sequence:

```
DD_bbbb
dd_bbbb
bbbb
```

Windows systems are case-insensitive with regard to environment variables, so there is no difference between the first two when using a GnuCOBOL implementation built for either Windows/MinGW or native Windows.

If an environment variable was found, its value will serve as the path and filename to the data file.

I. If no environment variable was found, or the *configuration file* (see [Compiler Configuration Files], page 649) used to compile the program had a `filename-mapping` value of `NO`, then the File Locator String value will serve as the path and filename to the data file.

J. Paths and file names may be specified on an absolute (`C:\\Data\\datafile.dat`, `/Data/datafile.dat`, ...) or relative to the current directory (`Data\\datafile.dat`, `Data/datafile.dat`, ...) basis. If no directory name is included (`datafile.dat`), the file must be in the current directory.

7. The `FILE STATUS` or `SORT STATUS` clause (they are both equivalent and only one or the other, if any, should be specified) is used to specify the name of a two-digit numeric data item into which an I/O status code will be saved after every I/O verb that is executed against the file. This does not actually allocate the data item — you must define the item yourself somewhere in the data division. Note that the following list is not definitive: more can be added and any tests should include one for non zeros as a catch all. Possible status codes that can be returned to a `FILE STATUS` data item are as follows:

`00`	Success
`02`	Success (Duplicate Record Key Written)
`04`	Success (Incomplete)
`05`	Success (Optional File Not Found)
`07`	Success (No Unit)
`10`	End of file reached if reading forward or beginning-of-file reached if reading backward
`14`	Out of key range

21	Key invalid
22	Key already exists
23	Key not found
24	Key boundary violation
30	Permanent I/O error
31	Inconsistent filename
34	Boundary violation
35	File not found
37	Permission denied
38	Closed with lock
39	Conflicting attribute
41	File already open
42	File not open
43	Read not done
44	Record overflow
46	Read error
47	`OPEN INPUT` denied (insufficient permissions to read file)
48	`OPEN OUTPUT` denied (insufficient permissions to write to file)
49	`OPEN I-O` denied (insufficient permissions to read and/or write file)
51	Record locked
52	End of page
57	`LINAGE` bad specification (I-O linage)
61	File sharing failure
71	Bad character
91	File not available

8. The `SHARING` clause defines the conditions under which the program will be willing (or not) to allow other programs executing at the same time to access the file. See [File Sharing], page 57, for the details.
9. The `LOCK` clause defines how concurrent access to the file will be managed on a record-by-record basis. See [Record Locking], page 59, for the details.
10. For syntax details for the `ORGANIZATION` clause, see next group of paragraphs.
11. A `SELECT` statement without an `ORGANIZATION` explicitly coded will be handled as if the following ORGANIZATION clause had been specified:

```
ORGANIZATION IS SEQUENTIAL
ACCESS MODE IS SEQUENTIAL
```

5.2.1.1. ORGANIZATION SEQUENTIAL

ORGANIZATION SEQUENTIAL Clause Syntax

```
[ ORGANIZATION|ORGANISATION IS ] RECORD BINARY SEQUENTIAL
  ~~~~~~~~~~~~ ~~~~~~~~~~~~                   ~~~~~~~~~~

   [ ACCESS MODE IS SEQUENTIAL ]
     ~~~~~~      ~~~~~~~~~~
```

Files declared as `ORGANIZATION SEQUENTIAL` will consist of records with no explicit end-of-record delimiter character sequences; records in such files are "delineated" by a calculated byte-offset (based on the maximum record length) into the file.

1. The reserved words `BINARY`, `IS`, `MODE` and `RECORD` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved words `ORGANIZATION` and `ORGANISATION` are interchangeable.
3. The phrase `ORGANIZATION IS` (and its internationalized alternative, `ORGANISATION IS`) is optional to provide compatibility with those (few) COBOL implementations that consider `ORGANIZATION` to be optional. Most COBOL implementations do require the word `ORGANIZATION`, so it should be used in new programs.
4. These files cannot be prepared with any standard text-editing or word processing software as all such programs will embed delimiter characters at the end of records (use `ORGANIZATION IS LINE SEQUENTIAL` instead).
5. These files may contain either `USAGE DISPLAY` or `USAGE COMPUTATIONAL` (of any variety) data since no binary data sequence can be accidentally interpreted as an end-of-record delimiter.
6. While records in a `ORGANIZATION SEQUENTIAL` file may be defined as having variable-length records, the file will be structured in such a manner as to reserve space for each record equal to the size of the largest possible record, based on the file's description in the `FILE SECTION`.
7. The `ACCESS MODE SEQUENTIAL` clause is optional because, if absent, it will be assumed anyway for this type of file. The internal structure of these files is such that they can only be processed in a sequential manner; in order to read the 100th record in such a file, for example, you first must read records 1 through 99.
8. Sequential files are processed using the following statements:
   - `CLOSE` (see [CLOSE], page 299)
   - `COMMIT` (see [COMMIT], page 300)
   - `DELETE` (see [DELETE], page 304)
   - `MERGE` (see [MERGE], page 349)
   - `OPEN` (see [OPEN], page 358)
   - `READ` (see [READ], page 366)
   - `REWRITE` (see [REWRITE], page 374)

- `SORT` (see [SORT], page 391)
- `UNLOCK` (see [UNLOCK], page 412)
- `WRITE` (see [WRITE], page 417)

5.2.1.2. ORGANIZATION LINE SEQUENTIAL

ORGANIZATION LINE SEQUENTIAL Clause Syntax

```
[ ORGANIZATION|ORGANISATION IS ] LINE SEQUENTIAL
  ~~~~~~~~~~~~ ~~~~~~~~~~~~      ~~~~ ~~~~~~~~~~

   [ ACCESS MODE IS SEQUENTIAL ]
     ~~~~~~      ~~~~~~~~~~

   [ PADDING CHARACTER IS literal-1 | identifier-1 ]
     ~~~~~~~
```

The `PADDING CHARACTER` clause is syntactically recognized but is otherwise non-functional.

Files declared as `ORGANIZATION LINE SEQUENTIAL` will consist of records terminated by an end-of-record delimiter character or character sequence.

1. The reserved words `CHARACTER`, `IS` and `MODE` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved words `ORGANIZATION` and `ORGANISATION` are interchangeable.
3. The phrase `ORGANIZATION IS` (and its internationalized alternative, `ORGANISATION IS`) is optional to provide compatibility with those (few) COBOL implementations that consider that word to be optional. Most COBOL implementations do require the word `ORGANIZATION`, so it should be used in new programs.
4. This is the only `ORGANIZATION` valid for files that are assigned to the `PRINTER` device.
5. These files may be created with any standard text-editing or word processing software capable of writing text files. Such files should not contain any `USAGE COMPUTATIONAL` or `BINARY` (of any variety) data since such fields could accidentally contain byte sequences that could be interpreted as an end-of-record delimiter.
6. Both fixed- and variable-length record formats are supported.
7. The end-of-record delimiter sequence will be `X'0A'` (an ASCII line-feed character) or a `X'0D0A'` (an ASCII carriage-return + line-feed sequence). The former is used on Unix implementations of GnuCOBOL (including Windows/MinGW, Windows/Cygwin and OSX implementations) while the latter would be used with native Windows implementations.
8. When reading a `LINE SEQUENTIAL` file, records in excess of the size implied by the file's description in the `FILE SECTION` will be truncated while records shorter than that size will be padded to the right with `SPACES`.
9. The `ACCESS MODE SEQUENTIAL` clause is optional because, if absent, it will be assumed anyway for this type of file. The internal structure of these files is such that the data can only be processed in a sequential manner; in order to read the 100th record in such a file, for example, you first must read records 1 through 99.
10. Files assigned to `PRINTER` or `CONSOLE` should be specified as `ORGANIZATION LINE SEQUENTIAL`.

11. Line Sequential files are processed using the following statements:
    - `CLOSE` (see [CLOSE], page 299)
    - `COMMIT` (see [COMMIT], page 300)
    - `DELETE` (see [DELETE], page 304)
    - `MERGE` (see [MERGE], page 349)
    - `OPEN` (see [OPEN], page 358)
    - `READ` (see [READ], page 366)
    - `REWRITE` (see [REWRITE], page 374)
    - `SORT` (see [SORT], page 391)
    - `UNLOCK` (see [UNLOCK], page 412)
    - `WRITE` (see [WRITE], page 417)

5.2.1.3. ORGANIZATION RELATIVE

ORGANIZATION RELATIVE Clause Syntax

```
[ ORGANIZATION|ORGANISATION IS ] RELATIVE
  ~~~~~~~~~~~~ ~~~~~~~~~~~~      ~~~~~~~~
   [ ACCESS MODE IS { SEQUENTIAL } ]
     ~~~~~~         { ~~~~~~~~~~ }
                    { DYNAMIC    }
                    { ~~~~~~~    }
                    { RANDOM     }
                      ~~~~~~
   [ RELATIVE KEY IS identifier-1 ]
     ~~~~~~~~
```

These files are files with an internal organization such that records may be processed in a sequential manner based upon their physical location in the file or in a random manner by allowing records to be read, written or updated by specifying the relative record number in the file.

1. The reserved words `IS`, `KEY` and `MODE` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved words `ORGANIZATION` and `ORGANISATION` are interchangeable.
3. The phrase `ORGANIZATION IS` (and its internationalized alternative, `ORGANISATION IS`) is optional to provide compatibility with those (few) COBOL implementations that consider that word to be optional. Most COBOL implementations do require the word `ORGANIZATION`, so it should be used in new programs.
4. `ORGANIZATION RELATIVE` files cannot be assigned to the `CONSOLE`, `DISPLAY`, `LINE ADVANCING` or `PRINTER` devices.
5. The `RELATIVE KEY` clause is optional only if `ACCESS MODE SEQUENTIAL` is specified.
6. While an `ORGANIZATION RELATIVE` file may be defined as having variable-length records, the file will be structured in such a manner as to reserve space for each record equal to the size of the largest possible record as defined by the file's description in the `FILE SECTION`.
7. `ACCESS MODE SEQUENTIAL`, the default `ACCESS MODE` if none is specified, indicates that the records of the file will be processed in a sequential manner, according to their physical sequence in the file.
8. `ACCESS MODE RANDOM` means that records will be processed in random sequence by specifying their record number in the file every time the file is read or written.
9. `ACCESS MODE DYNAMIC` indicates the program may switch back and forth between `SEQUENTIAL` and `RANDOM` mode during execution. The file starts out initially in `SEQUENTIAL` mode when first opened but the program may use the `START` statement (see [START], page 397) to switch between sequential and random access.

10. The `RELATIVE KEY` data item is a numeric data item that cannot be defined as a field within records of this file. Its purpose is to return the current relative record number of a relative file that is being processed in `SEQUENTIAL` access mode and to serve as a key that specifies the relative record number to be read or written when processing a relative file in `RANDOM` access mode.
11. Relative files are processed using the following statements:
    - `CLOSE` (see [CLOSE], page 299)
    - `COMMIT` (see [COMMIT], page 300)
    - `DELETE` (see [DELETE], page 304)
    - `MERGE` (see [MERGE], page 349), `ACCESS MODE RANDOM` not allowed
    - `OPEN` (see [OPEN], page 358)
    - `READ` (see [READ], page 366)
    - `REWRITE` (see [REWRITE], page 374)
    - `SORT` (see [SORT], page 391), `ACCESS MODE RANDOM` not allowed
    - `START` (see [START], page 397)
    - `UNLOCK` (see [UNLOCK], page 412)
    - `WRITE` (see [WRITE], page 417)

5.2.1.4. ORGANIZATION INDEXED

ORGANIZATION INDEXED Clause Syntax

```
[ ORGANIZATION|ORGANISATION IS ] INDEXED
  ~~~~~~~~~~~~ ~~~~~~~~~~~~      ~~~~~~~
   [ ACCESS MODE IS { SEQUENTIAL } ]
     ~~~~~~         { ~~~~~~~~~  }
                    { DYNAMIC    }
                    { ~~~~~~~    }
                    { RANDOM     }
                      ~~~~~~
   [ RECORD KEY IS { [ data-name-1       ]
     ~~~~~~
                   { [ record-key-name-1 ]
                     [ =|{SOURCE IS} data-name-2 ] ...  ]  }
                          ~~~~~~
   [ ALTERNATE RECORD KEY IS { [ data-name-3       ]
     ~~~~~~~~~ ~~~~~~
                             { [ record-key-name-2 ]
                               [ =|{SOURCE IS} data-name-4 ] ...  ]  }
                                    ~~~~~~
                             [ WITH DUPLICATES ] ]...
                                   ~~~~~~~~~~
                             [ SUPPRESS WHEN ALL literal     ]
                               ~~~~~~~~~~~~~~~~~
                             [ SUPPRESS WHEN SPACES | ZEROES ]
                               ~~~~~~~~~~~~~~~~~~~   ~~~~~~
```

Indexed files, like relative files, may have their records processed in either a sequential or random manner. Unlike relative files, however, the actual location of a record in an indexed file is calculated automatically based upon the value(s) of one or more alphanumeric fields within records of the file. For example, an indexed file containing product data might use the product identification code as a record key. This means you may read, write or update the A6G4328th record or the Z8X7723th record directly, based upon the product id value of those records!

1. The reserved words IS, KEY and MODE are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved words ORGANIZATION and ORGANISATION are interchangeable.
3. The phrase ORGANIZATION IS (and its internationalized alternative, ORGANISATION IS) is optional to provide compatibility with those (few) COBOL implementations that consider that word to be optional. Most COBOL implementations do require the word ORGANIZATION, so it should be used in new programs.

4. `ORGANIZATION INDEXED` files cannot be assigned to `CONSOLE`, `DISPLAY`, `KEYBOARD`, `LINE ADVANCING` or `PRINTER`.
5. `ACCESS MODE SEQUENTIAL`, the default `ACCESS MODE` if none is specified, indicates that the records of the file will be processed in a sequential manner with respect to the values of the `RECORD KEY` or the `ALTERNATE RECORD KEY` most-recently referenced on a `START` statement (see [START], page 397).
6. `ACCESS MODE RANDOM` means that records will be processed in random sequence by accessing the record with specific record key or alternate record key values.
7. `ACCESS MODE DYNAMIC` allows the file will be processed either in `RANDOM` or `SEQUENTIAL` mode; the program may switch between the two modes as needed. The `START` statement is used to make the switch between modes.
8. The `RECORD KEY` clause defines the field within the record used to provide the primary access to records within the file. No two records in the file will be allowed to have the same `PRIMARY KEY` field value. The `SOURCE IS` clause is for use with `Split Keys`.
9. The `ALTERNATE RECORD KEY` clause, if used, defines an additional field within the record that provides an alternate means of directly accessing records or an additional field by which the file's contents may be processed sequentially. You have the choice of allowing records to have duplicate alternate key values, if necessary.
10. There may be multiple `ALTERNATE RECORD KEY` clauses, each defining an additional alternate key for the file.
11. Usage of the `SUPPRESS WHEN` clause is used when `Sparse Keys` are required which may take the form for a literal or spaces or zeroes.
12. Indexed files are processed using the following statements:
    - `CLOSE` (see [CLOSE], page 299)
    - `COMMIT` (see [COMMIT], page 300)
    - `DELETE` (see [DELETE], page 304)
    - `MERGE` (see [MERGE], page 349), `ACCESS MODE RANDOM` not allowed
    - `OPEN` (see [OPEN], page 358)
    - `READ` (see [READ], page 366)
    - `REWRITE` (see [REWRITE], page 374)
    - `SORT` (see [SORT], page 391), `ACCESS MODE RANDOM` not allowed
    - `START` (see [START], page 397)
    - `UNLOCK` (see [UNLOCK], page 412)
    - `WRITE` (see [WRITE], page 417)

5.2.2. SAME RECORD AREA

I-O-CONTROL SAME AREA Syntax

```
SAME { SORT-MERGE } AREA FOR file-name-1...  .
     { SORT       }
     { RECORD     }
```

The `SAME SORT-MERGE` and `SAME SORT` clauses are syntactically recognized but are otherwise non-functional.

The `SAME RECORD AREA` clause allows you to specify that multiple files should share the same input and output memory buffers.

1. The reserved words `AREA` and `FOR` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. This statement must be terminated with a period.
3. While coding only a single file name (the repeated *file-name-1* item) is syntactically valid, this statement will have no effect upon the program unless at least two files are specified.
4. The effect of this statement will be to cause the specified files to share the same I/O buffer in memory. These buffers can sometimes get quite large, and by having multiple files share the same buffer memory you may significantly cut down the amount of memory the program is using (thus making "room" for more procedural code or data). If you do use this feature, take care to ensure that no more than one of the specified files are ever OPEN simultaneously.

5.2.3. MULTIPLE FILE

I-O-CONTROL MULTIPLE FILE Syntax

```
MULTIPLE FILE TAPE CONTAINS
~~~~~~~~
   { file-name-1 [ POSITION integer-1 ] }...
                 ~~~~~~~~

   .
```

The `MULTIPLE FILE TAPE` clause is obsolete and is therefore recognized but not functional.

End of Chapter 5 — ENVIRONMENT DIVISION

6. DATA DIVISION

DATA DIVISION Syntax

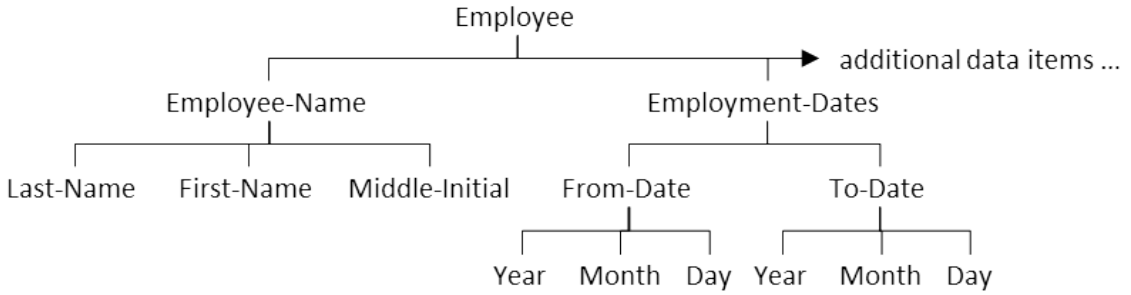
```
  DATA DIVISION.
  ~~~~ ~~~~~~~~
[ FILE SECTION.
  ~~~~ ~~~~~~~
  { File/Sort-Description [ { FILE-SECTION-Data-Item } ]...  }...  ]
  {                         { 01-Level-Constant      }       }
  {                         { 78-Level-Constant      }       }
  { 01-Level-Constant                                        }
  { 78-Level-Constant                                        }
[ WORKING-STORAGE SECTION.
  ~~~~~~~~~~~~~~~ ~~~~~~~
  [ { WORKING-STORAGE-SECTION-Data-Item } ]...  ]
    { 01-Level-Constant                 }
    { 78-Level-Constant                 }
[ LOCAL-STORAGE SECTION.
  ~~~~~~~~~~~~~ ~~~~~~~
  [ { LOCAL-STORAGE-SECTION-Data-Item } ]...  ]
    { 01-Level-Constant               }
    { 78-Level-Constant               }
[ LINKAGE SECTION.
  ~~~~~~~ ~~~~~~~
  [ { LINKAGE-SECTION-Data-Item } ]...  ]
    { 01-Level-Constant         }
    { 78-Level-Constant         }
[ REPORT SECTION.
  ~~~~~~ ~~~~~~~
  { Report-Description [ { Report-Group-Definition } ]...  }...  ]
  {                      { 01-Level-Constant       }       }
  {                      { 78-Level-Constant       }       }
  { 01-Level-Constant                                      }
  { 78-Level-Constant                                      }
[ SCREEN SECTION.
  ~~~~~~ ~~~~~~~
  [ { SCREEN-SECTION-Data-Item } ]...  ]
    { 01-Level-Constant        }
    { 78-Level-Constant        }
```

All data used by any COBOL program must be defined in one of the six sections of the data division, depending upon the purpose of the data.

1. If no data will be described in one of the data division sections, that section header may be omitted.
2. If no data division sections are needed, the `DATA DIVISION.` header itself may be omitted.
3. If more than one section is needed in the data division (a common situation), the sections must be coded in the sequence they are presented above.

6.1. Data Definition Principles

GnuCOBOL data items, like those of other COBOL implementations, are described in a hierarchical manner. This accommodates the fact that data items frequently need to be able to be broken up into subordinate items. Take for example, the following logical layout of a portion of a data item named `Employee`:



The `Employee` data item consists of two subordinate data items — an `Employee-Name` and an `Employment-Dates` data item (presumably there would be a lot of others too, but we don't care about them right now). As the diagram shows, each of those data items are, in turn, broken down into subordinate data items. This hierarchy of data items can get rather `deep`, and GnuCOBOL, like other COBOL implementations, can handle up to 49 levels of such hierarchical structures.

As was presented earlier (see [Structured Data], page 14), a data item that is broken down into other data items is referred to as a group item, while one that isn't broken down is called an elementary item.

COBOL uses the concept of a *level* number to indicate the level at which a data item occurs in a data structure such as the example shown above. When these data items are defined, they are all defined together with a number in the range 1-49 specified in front of their names. Over the years, a convention has come to exist among COBOL programmers that level numbers are always coded as two-digit numbers — they don't *need* to be specified as two-digit numbers, but every example you see in this document will take that approach!

The data item at the top, also referred to as a *record*, always has a level number of 01. After that, you may assign level numbers as you wish (01–02–03–04. . . , 01–05–10–15. . . , etc.), as long as you follow these simple rules:

1. Every data item at the same *level* of a hierarchy diagram such as the one you see here (if you were to make one, which you rarely will, if ever, once you get used to this concept) must have the same level number.

2. Every new level uses a level number that is strictly greater than the one used in the parent (next higher) level.
3. When describing data hierarchies, you may never use a level number greater than 49 (except for 66, 77, 78 and 88 which have very special meanings (see [Special Data Items], page 146).

So, the definition of these data items in a GnuCOBOL program would go something like this:

```
01  Employee
    05 Employee-Name
       10 Last-Name
       10 First-Name
       10 Middle-Initial
    05 Employment-Dates
       10 From-Date
          15 Year
          15 Month
          15 Day
       10 To-Date
          15 Year
          15 Month
          15 Day
```

The indentation is purely at the discretion of the programmer to make things easier for humans to read (the compiler couldn't care less). Historically, COBOL implementations that required Fixed Format Mode source programs required that the `01` level number begin in Area A and that everything else begins in Area B. GnuCOBOL only requires that all data definition syntax occur in columns 8-72. In Free Format Mode, of course, there aren't even those limitations.

Did you notice that there are two each of `Year`, `Month` and `Day` data names defined? That's perfectly legal, provided that each can be uniquely `qualified` so as to be distinct from the other. Take for example the `Year` items. One is defined as part of the `From-Date` data item while the other is defined as part of the `To-Date` data item. In COBOL, we would actually code references to these two data items as either `Year OF From-Date` and `Year OF To-Date` or `Year IN From-Date` and `Year IN To-Date` (COBOL allows either `IN` or `OF` to be used). Since these references would clarify any confusion to us as to which `Year` might be referenced, the GnuCOBOL compiler won't be confused either.

The coding example shown above is incomplete; it only describes the data item names and their hierarchical relationships to one other. In addition, any valid data item definitions will also need to describe what type of data is to be contained in a data item (Numeric? Alphanumeric? Alphabetic?), how much data can be held in the data item and a multitude of other characteristics.

When group items are being defined, subordinate items may be assigned the "name" `FILLER`. There may be any number of `FILLER` items defined within a group item. A data item named `FILLER` cannot be referenced directly; these items are generally used to specify an unused portion of the total storage allocated to a group item. Note that it is possible

that the name of the group item itself might be specified as `FILLER` if there is no need to ever refer directly to the group structure itself.

6.2. FILE SECTION

FILE SECTION Syntax

```
[ FILE SECTION.
  ~~~~ ~~~~~~~
  { File/Sort-Description [ { FILE-SECTION-Data-Item } ]...  }...  ]
  {                         { 01-Level-Constant      }      }
  {                         { 78-Level-Constant      }      }
  { 01-Level-Constant                                       }
  { 78-Level-Constant                                       }
```

Every file that has been referenced by a `SELECT` statement (see [SELECT], page 104) must also be described in the file section of the data division.

Files destined for use as sort/merge work files must be described with a Sort/Merge File Description (`SD`) while every other file is described with a File Description (`FD`). Each of these descriptions will almost always be followed with at least one record description.

6.2.1. File/Sort-Description

File/Sort-Description Syntax

```
FD|SD file-name-1 [ IS EXTERNAL|GLOBAL ]
~~ ~~               ~~~~~~~~ ~~~~~~
[ BLOCK CONTAINS [ integer-1 TO ] integer-2 CHARACTERS|RECORDS ]
  ~~~~~          ~~             ~~~~~~~~~~ ~~~~~~~
[ CODE-SET IS alphabet-name-1 ]
  ~~~~~~~~
[ DATA { RECORD IS   } identifier-1...  ]
  ~~~~ { ~~~~~~      }
       { RECORDS ARE }
         ~~~~~~~
[ LABEL { RECORD IS   } OMITTED|STANDARD ]
  ~~~~~ { ~~~~~~      } ~~~~~~~ ~~~~~~~~
        { RECORDS ARE }
          ~~~~~~~
[ LINAGE IS integer-3 | identifier-2 LINES
  ~~~~~~
    [ LINES AT BOTTOM integer-4 | identifier-3 ]
              ~~~~~~
    [ LINES AT TOP integer-5 | identifier-4 ]
              ~~~
    [ WITH FOOTING AT integer-6 | identifier-5 ] ]
           ~~~~~~~
[ RECORD { CONTAINS [ integer-7 TO ] integer-8 CHARACTERS   } ]
  ~~~~~~ {                      ~~                         }
         { IS VARYING IN SIZE                              }
         {    ~~~~~~~                                      }
         {     [ FROM [ integer-7 TO ] integer-8 CHARACTERS }
         {                       ~~                       }
         {         DEPENDING ON identifier-6 ]             }
                   ~~~~~~~~~
[ RECORDING MODE IS recording-mode ]
  ~~~~~~~~~
[ { REPORT IS   } report-name-1...  ]
  { ~~~~~~      }
  { REPORTS ARE }
    ~~~~~~~
[ VALUE OF implementor-name-1 IS literal-1 | identifier-7 ] .
  ~~~~~ ~~
```

The `BLOCK CONTAINS`, `DATA RECORD`, `LABEL RECORD`, `RECORDING MODE` and `VALUE OF` clauses are syntactically recognized but are obsolete and non-functional. These clauses should not be coded in new programs.

1. The reserved words `ARE`, `AT`, `CHARACTERS` (`RECORD` clause only), `CONTAINS`, `FROM`, `IN`, `IS`, `ON` and `WITH` are optional and may be included, or not, at the discretion of the programmer. The presence or absence of these words has no effect upon the program.
2. The terms `RECORD IS` and `RECORDS ARE` are interchangeable.
3. The terms `REPORT IS` and `REPORTS ARE` are interchangeable.
4. Only files intended for use as work files for either the `SORT` (see [SORT], page 391) or `MERGE` (see [MERGE], page 349) statements should be coded with an SD — all others should be defined with a FD.
5. The sequence in which files are defined via `FD` or `SD`, as compared to the sequence in which their `SELECT` statements were coded, is irrelevant.
6. The name specified as *file-name-1* must exactly match the name specified on the file's `SELECT` statement.
7. The `CODE-SET` clause allows a custom alphabet, defined in the `SPECIAL-NAMES` (see [SPECIAL-NAMES], page 92) paragraph, to be associated with a file. This clause is valid only when used with sequential or line sequential files and usage can depend of the compiler version in use, platform used and other factors, see the NEWS file supplied with your version of the compiler for more details.
8. The `LINAGE` clause may only be specified in the `FD` of a sequential or line sequential file. If used with a sequential file, the organization of that file will be implicitly changed to line sequential. The various components of the `LINAGE` clause define the layout of printed pages as follows:

   LINES AT TOP
   : Number of unused (*i.e.* left blank) lines at the top of every page. The default if this if not specified is zero.

   LINES AT BOTTOM
   : Number of unused (*i.e.* left blank) lines at the bottom of every page. The default if this if not specified is zero.

   LINAGE IS n LINES
   : Total number of used/usable lines on the page.

   The sum of the previous three specifications should be the total number of possible lines available on one printed page.

   FOOTING AT
   : Line number beyond which nothing may be printed except for any footing that is to appear on every page. The default for this if not specified is zero, meaning there will be no footings. This value cannot be larger than the `LINAGE IS` *n* `LINES` value.

9. This page structure — once defined — can be automatically enforced by the `WRITE` statement (see [WRITE], page 417).
10. Specifying a `LINAGE` clause in an `FD` will cause the `LINAGE-COUNTER` special register to be created for the file. This automatically-created data item will always contain the

current relative line number on the page being prepared which will serve as the starting point for a `WRITE` statement.

11. The `RECORD CONTAINS` and `RECORD IS VARYING` clauses are ignored (with a warning message issued) when used with line sequential files. With other file organizations, these mutually-exclusive clauses define the length of data records within the file. The data item specified as *identifier-6* must be defined within one of the record descriptions of *file-name-1*.
12. The `REPORT IS` clause announces to the compiler that the file will be dedicated to the Report Writer Control System (RWCS); the clause names one or more reports, each to be described in the report section. The following special rules apply when the `REPORT` clause is used:
    A. The clause may only be specified in the `FD` of a sequential or line sequential file. If used with a sequential file, the organization of that file will be implicitly changed to line sequential.
    B. The `FD` cannot be followed by record descriptions. Detailed descriptions of data to be printed to the file will be defined in the `REPORT SECTION` (see [REPORT SECTION], page 135).
    C. If a `LINAGE` clause is also specified, Values specified for `LINAGE IS` and `FOOTING AT` will be ignored. The values of `LINES AT BOTTOM` and `LINES AT TOP`, if any, will be honoured.
13. The following special rules apply only to sort/merge work files:
    A. Sort/merge work files should be assigned to `DISK` (or `DISC`) on their `SELECT` statements.
    B. Sorts and merges will be performed in memory, if the amount of data being sorted allows.
    C. Should actual disk work files be necessary due to the amount of data being sorted or merged, they will be automatically allocated to disk in a folder defined by:
       - The `TMPDIR` run-time environment variable (see [Run Time Environment Variables], page 658)
       - The `TMP` run-time environment variable
       - The `TEMP` run-time environment variable

       (in that order).
    D. These disk files will be automatically purged upon `SORT` or `MERGE` termination. They will also be purged if the program terminates abnormally before the `SORT` or `MERGE` finishes. Should you ever need to know, temporary sort/merge work files will be named `cob*.tmp`.
    E. If you specify a specific filename in the sort/merge work file's `SELECT`, it will be ignored.
14. See [Data Description Clauses], page 154, for information on the `EXTERNAL` and `GLOBAL` options.

6.2.2. FILE-SECTION-Data-Item

FILE-SECTION-Data-Item Syntax

```
level-number [ identifier-1 | FILLER ] [ IS GLOBAL|EXTERNAL ]
                              ~~~~~~         ~~~~~~ ~~~~~~~~

[ BLANK WHEN ZERO ]
  ~~~~~      ~~~~

[ JUSTIFIED RIGHT ]
  ~~~~

[ OCCURS [ integer-1 TO ] integer-2 TIMES
  ~~~~~~              ~~   UNBOUNDED
                           ~~~~~~~~~

       [ DEPENDING ON identifier-2 ]
         ~~~~~~~~~

       [ STEP identifier-6 ]
       [ ASCENDING|DESCENDING KEY IS identifier-3 ]
         ~~~~~~~~~ ~~~~~~~~~~

       [ INDEXED BY identifier-4 ] ]
         ~~~~~~~

[ PICTURE IS picture-string ]
  ~~~

[ REDEFINES identifier-5 ]
  ~~~~~~~~~

[ SIGN IS LEADING|TRAILING [ SEPARATE [CHARACTER] ] ]
  ~~~~    ~~~~~~~ ~~~~~~~~   ~~~~~~~~

[ SYNCRONIZED|SYNCHRONISED [ LEFT|RIGHT ] ]
  ~~~~        ~~~~           ~~~~ ~~~~~

[ USAGE IS data-item-usage ] .  [ FILE-SECTION-Data-Item ]...
  ~~~~~
```

The `LEFT` and `RIGHT` (SYNCRONIZED) clauses are syntactically recognized but are otherwise non-functional.

Every sort file description (`SD` or `FD`) must be followed by at least one 01-level data item, except for file descriptions containing the `REPORT IS` clause. These 01-level data items, in turn, may be broken down into subordinate group and elementary items. An 01-level data item defined here in the file section is also known as a *Record*, even if it is an elementary item, provided that elementary item lacks the `CONSTANT` attribute.

1. The reserved words `BY`, `IS`, `KEY`, `ON` and `WHEN` are optional and may be included, or not, at the discretion of the programmer. The presence or absence of these words has no effect upon the program.
2. The reserved words `SYNCRONIZED` and `SYNCRONIZED` are interchangeable. Both may be abbreviated to `SYNC`.

3. The reserved word `PICTURE` may be abbreviated to `PIC`.
4. As the syntax diagram shows, the definition of a *FILE-SECTION-Data-Item* is a recursive one in that there may be any number of such specifications coded following a FD or SD. The first such specification must have a level number of 01, and will describe a specific format of data record within the file. Specifications that follow that one may have level numbers greater than 01, in which case they are defining a hierarchical breakdown of the record. The definition of a record is terminated when one of the following occurs:

   Another 01-level item is found
   : signifies the start of another record layout for the file.

   Another `FD` or `SD` is found
   : marks the completion of the detailed description of the file and begins another.

   A division or section header is found
   : also marks the completion of the detailed description of the file and signifies the end of the file section as well.
5. Every FILE-SECTION-Data-Item description must be terminated with a period.
6. If there are multiple record descriptions present for a given `FD` or `SD`, the one with the longest length will define the size of the record buffer into which a `READ` statement (see [READ], page 366) or a `RETURN` statement (see [RETURN], page 373) will deliver data read from the file and from which a `WRITE` statement (see [WRITE], page 417) or `RELEASE` statement (see [RELEASE], page 371) statement will obtain the data to be written to the file.
7. The various 01-level record descriptions for a file description implicitly share that one common record buffer (thus, they provide different ways to view the structure of data that can exist within the file). Record buffers can be shared between files by using the `SAME RECORD AREA` (see [SAME RECORD AREA], page 117) clause.
8. The only valid level numbers are 01-49, 66, 77, 78 and 88. Level numbers 66, 77, 78 and 88 all have special uses — See [Special Data Items], page 146, for details.
9. Not specifying an *identifier-1* or `FILLER` immediately after the level number has the same effect as if `FILLER` were specified. A data item named `FILLER` cannot be referenced directly; these items are generally used to specify an unused portion of the total storage allocated to a group item or to describe a group item whose contents which will only be referenced using the names of those items that belong to it.
10. `EXTERNAL` cannot be combined with `GLOBAL` or `REDEFINES`.
11. File section data buffers (and therefore all 01-level record layouts defined in the file section) are initialized to all binary zeros when the program is loaded into storage.
12. See [Data Description Clauses], page 154, for information on the usage of the various data description clauses.

6.3. WORKING-STORAGE SECTION

WORKING-STORAGE-SECTION-Data-Item Syntax

```
level-number [ identifier-1 | FILLER ] [ IS GLOBAL | EXTERNAL ]
                              ~~~~~~        ~~~~~~   ~~~~~~~~

[ BASED ]
  ~~~~~

[ BLANK WHEN ZERO ]
  ~~~~~      ~~~~

[ JUSTIFIED RIGHT ]
  ~~~~

[ OCCURS [ integer-1 TO ] integer-2 TIMES
  ~~~~~~                 ~~   UNBOUNDED
                              ~~~~~~~~~

      [ DEPENDING ON identifier-2 ]
        ~~~~~~~~~

      [ ASCENDING|DESCENDING KEY IS identifier-3 ]
        ~~~~~~~~~ ~~~~~~~~~~

      [ INDEXED BY identifier-4 ] ]
        ~~~~~~~

[ PICTURE IS picture-string ]
  ~~~

[ REDEFINES identifier-5 ]
  ~~~~~~~~~

[ SIGN IS LEADING|TRAILING [ SEPARATE CHARACTER ] ]
  ~~~~    ~~~~~~~ ~~~~~~~~   ~~~~~~~~

[ SYNCRONIZED|SYNCHRONISED [ LEFT|RIGHT ] ]
  ~~~~        ~~~~           ~~~~ ~~~~~

[ USAGE IS data-item-usage ]
  ~~~~~

[ VALUE IS [ ALL ] literal-1 ] .  [ WORKING-STORAGE-SECTION-Data-Item ]...
  ~~~~~      ~~~
```

The `LEFT` and `RIGHT` (SYNCRONIZED) clauses are syntactically recognized but are otherwise non-functional.

The working-storage section is used to describe data items that are not part of files, screens or reports and whose data values persist throughout the execution of the program.

1. The reserved words `BY`, `CHARACTER`, `IS`, `KEY`, `ON`, `RIGHT` (JUSTIFIED), `TIMES` and `WHEN` are optional and may be included, or not, at the discretion of the programmer. The presence or absence of these words has no effect upon the program.
2. The reserved words `SYNCRONIZED` and `SYNCHRONISED` are interchangeable. Both may be abbreviated as `SYNC`.

3. The reserved word `PICTURE` may be abbreviated to `PIC`.
4. The reserved word `JUSTIFIED` may be abbreviated to `JUST`.
5. As the syntax diagram shows, the definition of a *`WORKING-STORAGE-SECTION-Data-Item`* is a recursive one in that there may be any number of such specifications coded following one another. The first such specification must have a level number of 01. Specifications that follow that one may have level numbers greater than 01, in which case they are defining a hierarchical breakdown of a record. The definition of a record is terminated when one of the following occurs:
   - Another 01-level item is found — this signifies the end of the definition of one record and the start of a another.
   - A 77-level item is found — this signifies the end of the definition of the record and begins the definition of a special data item; See [77-Level Data Items], page 151, for more information.
   - A division or section header is found — this also marks the completion of a record and signifies the end of the working-storage section as well.
6. Every *`WORKING-STORAGE-SECTION-Data-Item`* description must be terminated with a period.
7. The only valid level numbers are 01-49, 66, 77, 78 and 88. Level numbers 01 through 49 are used to define data items that may be part of a hierarchical structure. Level number 01 can also be used to define a constant — an item with an unchangeable value specified at compilation time.
8. Level numbers 66, 77, 78 and 88 all have special uses — See [Special Data Items], page 146, for details.
9. Not specifying an *identifier-1* or `FILLER` immediately after the level number has the same effect as if `FILLER` were specified. A data item named `FILLER` cannot be referenced directly; these items are generally used to specify an unused portion of the total storage allocated to a group item or to describe a group item whose contents which will only be referenced using the names of those items that belong to it.
10. Data items defined within the working-storage section are automatically initialized once — as the program in which the data is defined is loaded into memory. Subprograms may be loaded into memory more than once (see the `CANCEL` statement (see [CANCEL], page 298)), in which case initialization will happen each time they are loaded. See [Data Initialization], page 26, for a discussion of the initialization rules.
11. See [Data Description Clauses], page 154, for information on the usage of the various data description clauses.

6.4. LOCAL-STORAGE SECTION

LOCAL-STORAGE-SECTION-Data-Item Syntax

```
level-number [ identifier-1 | FILLER ] [ IS GLOBAL|EXTERNAL ]
                              ~~~~~~         ~~~~~~ ~~~~~~~~

[ BASED ]
  ~~~~~

[ BLANK WHEN ZERO ]
  ~~~~~      ~~~~

[ JUSTIFIED RIGHT ]
  ~~~~

[ OCCURS [ integer-1 TO ] integer-2 TIMES
  ~~~~~~                 ~~   UNBOUNDED
                              ~~~~~~~~~

      [ DEPENDING ON identifier-2 ]
        ~~~~~~~~~

      [ ASCENDING|DESCENDING KEY IS identifier-3 ]
        ~~~~~~~~~ ~~~~~~~~~~

      [ INDEXED BY identifier-4 ] ]
        ~~~~~~~

[ PICTURE IS picture-string ]
  ~~~

[ REDEFINES identifier-5 ]
  ~~~~~~~~~

[ SIGN IS LEADING|TRAILING [ SEPARATE CHARACTER ] ]
  ~~~~    ~~~~~~~ ~~~~~~~~   ~~~~~~~~

[ SYNCRONIZED|SYNCHRONISED [ LEFT|RIGHT ] ]
  ~~~~         ~~~~           ~~~~ ~~~~~

[ USAGE IS data-item-usage ]
  ~~~~~

[ VALUE IS [ ALL ] literal-1 ] .  [ LOCAL-STORAGE-SECTION-Data-Item ]...
  ~~~~~      ~~~
```

The `LEFT` and `RIGHT` (SYNCRONIZED) clauses are syntactically recognized but are otherwise non-functional.

The local-storage section is similar to working-storage, but describes data within a subprogram that will be dynamically allocated and initialized (automatically) each time the subprogram is executed. See [Data Initialization], page 26, for the rules of data initialization.

1. The reserved words `BY`, `CHARACTER IS`, `KEY`, `ON`, `RIGHT` (JUSTIFIED), `TIMES` and `WHEN` are optional and may be included, or not, at the discretion of the programmer. The presence or absence of these words has no effect upon the program.

2. The reserved words `SYNCRONIZED` and `SYNCHRONISED` are interchangeable. Both may be abbreviated as `SYNC`.
3. The reserved word `PICTURE` may be abbreviated to `PIC`.
4. The reserved word `JUSTIFIED` may be abbreviated to `JUST`.
5. As the syntax diagram shows, the definition of a *`LOCAL-STORAGE-SECTION-Data-Item`* is a recursive one in that there may be any number of such specifications coded following one another. The first such specification must have a level number of 01. Specifications that follow that one may have level numbers greater than 01, in which case they are defining a hierarchical breakdown of a record. The definition of a record is terminated when one of the following occurs:
   - Another 01-level item is found — this signifies the end of the definition of one record and the start of a another.
   - A division or section header is found — this also marks the completion of a record and signifies the end of the local-storage section as well.
6. Every *`LOCAL-STORAGE-SECTION-Data-Item`* description must be terminated with a period.
7. The only valid level numbers are 01-49, 66, 77, 78 and 88. Level numbers 01 through 49 are used to define data items that may be part of a hierarchical structure. Level number 01 can also be used to define a constant — an item with an unchangeable value specified at compilation time.
8. Level numbers 66, 77, 78 and 88 all have special uses — See [Special Data Items], page 146, for details.
9. Not specifying an *identifier-1* or `FILLER` immediately after the level number has the same effect as if `FILLER` were specified. A data item named `FILLER` cannot be referenced directly; these items are generally used to specify an unused portion of the total storage allocated to a group item or to describe a group item whose contents which will only be referenced using the names of those items that belong to it.
10. Local-storage cannot be used in nested subprograms.
11. See [Data Description Clauses], page 154, for information on the usage of the various data description clauses.

6.5. LINKAGE SECTION

LINKAGE-SECTION-Data-Item Syntax

```
level-number [ identifier-1 | FILLER ] [ IS GLOBAL|EXTERNAL ]
                              ~~~~~~        ~~~~~~ ~~~~~~~~

[ ANY LENGTH ]
  ~~~ ~~~~~~

[ BASED ]
  ~~~~~

[ BLANK WHEN ZERO ]
  ~~~~~      ~~~~

[ JUSTIFIED RIGHT ]
  ~~~~

[ OCCURS [ integer-1 TO ] integer-2 TIMES
  ~~~~~~             ~~

      [ DEPENDING ON identifier-3 ]
        ~~~~~~~~~

      [ ASCENDING|DESCENDING KEY IS identifier-4 ]
        ~~~~~~~~~ ~~~~~~~~~~

      [ INDEXED BY identifier-5 ] ]
        ~~~~~~~

[ PICTURE IS picture-string ]
  ~~~

[ REDEFINES identifier-6 ]
  ~~~~~~~~~

[ SIGN IS LEADING|TRAILING [ SEPARATE CHARACTER ] ]
  ~~~~    ~~~~~~~ ~~~~~~~~   ~~~~~~~~

[ SYNCRONIZED|SYNCHRONISED [ LEFT|RIGHT ] ]
  ~~~~         ~~~~           ~~~~ ~~~~~

[ USAGE IS data-item-usage ] .  [ LINKAGE-SECTION-Data-Item ]...
  ~~~~~
```

The `LEFT` and `RIGHT` (SYNCRONIZED) clauses are syntactically recognized but are otherwise non-functional.

The linkage section describes data within a subprogram that serves as either input arguments to or output results from the subprogram.

1. The reserved words `BY`, `CHARACTER`, `IS`, `KEY`, `ON` and `WHEN` are optional and may be included, or not, at the discretion of the programmer. The presence or absence of these words has no effect upon the program.
2. The reserved words `SYNCRONIZED` and "`SYNCHRONISED`" are interchangeable. Both may be abbreviated as `SYNC`.
3. The reserved word `PICTURE` may be abbreviated to `PIC`.

4. The reserved word `JUSTIFIED` may be abbreviated to `JUST`.
5. As the syntax diagram shows, the definition of a *`LINKAGE-SECTION-Data-Item`* is a recursive one in that there may be any number of such specifications coded following one another. The first such specification must have a level number of 01. Specifications that follow that one may have level numbers greater than 01, in which case they are defining a hierarchical breakdown of a record. The definition of a record is terminated when one of the following occurs:
   - Another 01-level item is found — this signifies the end of the definition of one record and the start of a another.
   - A division or section header is found — this also marks the completion of a record and signifies the end of the linkage section as well.
6. Every *`LINKAGE-SECTION-Data-Item`* description must be terminated with a period.
7. The only valid level numbers are 01-49, 66, 77, 78 and 88. Level numbers 01 through 49 are used to define data items that may be part of a hierarchical structure. Level number 01 can also be used to define a constant — an item with an unchangeable value specified at compilation time.
8. Level numbers 66, 77, 78 and 88 all have special uses — See [Special Data Items], page 146, for details.
9. It is expected that:
   A. A linkage section should occur only within a subprogram. The compiler will not prevent its use in a main program, however.
   B. All 01-level data items described within a subprogram's linkage section should appear in a `PROCEDURE DIVISION USING` (see [PROCEDURE DIVISION USING], page 248) or as arguments on an `ENTRY` statement.
   C. Each 01-level data item described within a subprogram's linkage section should correspond to an argument passed on a `CALL` statement (see [CALL], page 293) or an argument on a function call to the subprogram.
10. Not specifying an *identifier-1* or `FILLER` immediately after the level number has the same effect as if `FILLER` were specified. A data item named `FILLER` cannot be referenced directly; these items are generally used to specify an unused portion of the total storage allocated to a group item or to describe a group item whose contents which will only be referenced using the names of those items that belong to it. In the linkage section, 01-level data items cannot be named `FILLER`.
11. No storage is allocated for data defined in the linkage section; the data descriptions there are merely defining storage areas that will be passed to the subprogram by a calling program. Therefore, any discussion of the default initialization of such data is irrelevant. It *is* possible, however, to manually allocate linkage section data items that aren't subprogram arguments via the `ALLOCATE` statement (see [ALLOCATE], page 290) statement. In such cases, initialization will take place as per the documentation of that statement.
12. See [Data Description Clauses], page 154, for information on the usage of the various data description clauses.

6.6. REPORT SECTION

REPORT SECTION Syntax

```
[ REPORT SECTION.
  ~~~~~~ ~~~~~~~
  { Report-Description [ { Report-Group-Definition } ]...  }...  ]
  {                      { 01-Level-Constant       }      }
  {                      { 78-Level-Constant       }      }
  { 01-Level-Constant                                     }
  { 78-Level-Constant                                     }
```

Report-Description (RD) Syntax

```
RD report-name [ IS GLOBAL ]
~~                 ~~~~~~
[ CODE IS literal-1 | identifier-1 ]
  ~~~~
[ { CONTROL IS   } { FINAL        }...  ]
  { ~~~~~~~      } { ~~~~~        }
  { CONTROLS ARE } { identifier-2 }
    ~~~~~~~~
[ PAGE [ { LIMIT IS   } ] [ { literal-2    } LINES ]
  ~~~~   { ~~~~~      }     { identifier-3 } ~~~~
         { LIMITS ARE }
           ~~~~~~
      [ literal-3 | identifier-4 COLUMNS|COLS ]
                                 ~~~~~~~ ~~~~
      [ HEADING IS literal-4 | identifier-5 ]
        ~~~~~~~
      [ FIRST DE|DETAIL IS literal-5 | identifier-6 ]
        ~~~~~ ~~ ~~~~~~
      [ LAST CH|{CONTROL HEADING} IS literal-6 | identifier-7 ]
        ~~~~ ~~  ~~~~~~~ ~~~~~~~
      [ LAST DE|DETAIL IS literal-7 | identifier-8 ]
        ~~~~ ~~ ~~~~~~
      [ FOOTING IS literal-8 | identifier-9 ] ] .
        ~~~~~~~
```

This section describes the layout of printed reports as well as many of the functional aspects of the generation of reports that will be produced via the Report Writer Control System. It is important to maintain the order of these clauses and ensure that all fields defined or

referenced with this section are actually defined in the WORKING-STORAGE SECTION and not elsewhere.

1. The reserved words ARE and IS are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The phrases CONTROL IS and CONTROLS ARE are interchangeable, as are the PAGE LIMIT and PAGE LIMITS phrases.
3. The reserved word LINES may be abbreviated as LINE.
4. The reserved word COLUMNS may be abbreviated as COLS.
5. Each report referenced on a REPORT IS clause (see [File/Sort-Description], page 124) must be described with a report description (RD).
6. See [GLOBAL], page 176, for information on the GLOBAL option.
7. Please see [Report Writer Features], page 25, if you have not read it already. It will familiarize you with the Report Writer terminology that follows.
8. The following rules pertain to the PAGE LIMITS clause:
   A. If no PAGE LIMITS clause is specified, the entire report will be generated as if it consists of a single arbitrarily long page.
   B. All literals (*literal-2* through *literal-8*) must be numeric with non-zero positive integer values.
   C. All identifiers (*identifier-2* through *identifier-8*) must be numeric, unedited with non-zero positive integer values.
   D. Any value specified for *literal-2* or *identifier-2* will define the total number of available lines on any report page, not counting any unused margins at the top and/or bottom of the page (defined by the LINES AT TOP and LINES AT BOTTOM values specified on the LINAGE clause of the FD this RD is linked to — see [File/Sort-Description], page 124).
   E. Any value specified for *literal-3* or *identifier-3* will be ignored.
   F. The HEADING clause defines the first line number at which a report heading or page heading may be presented.
   G. The FIRST DETAIL clause defines the first line at which a detail group may be presented.
   H. The LAST CONTROL HEADING clause defines the last line at which any line of a control heading may be presented.
   I. The LAST DETAIL clause defines the last line at which any line of a detail group may be presented.
   J. The FOOTING clause defines the last line at which any line of a control footing group may be presented.
   K. The following rules establish default values for the various PAGE LIMIT clauses, assuming there is one:

      HEADING default is one (1)

      FIRST DETAIL HEADING
      value is used

`LAST CONTROL HEADING`
: value from `LAST DETAIL` or, if that is absent, the value from `FOOTING` or, if that too is absent, the value from `PAGE LIMIT`

`LAST DETAIL`
: value from `FOOTING` or, if that is absent, the value from `PAGE LIMIT`

`FOOTING`
: value from `LAST DETAIL` or, if that is absent, the value from `PAGE LIMIT`

L. For the values specified on a `PAGE LIMIT` clause to be valid, all of the following must be true:

- `FIRST DETAIL` $\leq$ `HEADING`
- `LAST CONTROL HEADING` $\leq$ `FIRST DETAIL`
- `LAST DETAIL` $\leq$ `LAST CONTROL HEADING`
- `FOOTING` $\leq$ `LAST DETAIL`

9. The following rules pertain to the `CONTROL` clause:

A. If there is no `CONTROL` clause, the report will contain no control breaks; this implies that there can be no `CONTROL HEADING` or `CONTROL FOOTING` report groups defined for this `RD`.

B. Include the reserved word `FINAL` if you want to include a special control heading before the first detail line is generated (`CONTROL HEADING FINAL`) or after the last detail line is generated (`CONTROL FOOTING FINAL`).

C. If you specify `FINAL`, it must be the first control break named in the `RD`.

D. Any *identifier-9* specifications included on the `CONTROL` clause are referencing data names defined in any data division section except for the report section.

E. There must be a `CONTROL HEADING` and/or `CONTROL FOOTING` report group defined in the report section for each *identifier-9*.

F. At execution time:

- Each time a `GENERATE` statement (see [GENERATE], page 332) is executed against a detail report group defined for this `RD`, the RWCS will check the contents of each *identifier-2* data item; whenever an *identifier-9*'s value has changed since the previous `GENERATE`, a control break condition will be in effect for that *identifier-2*.
- Once the list of control breaks has been determined, the `CONTROL FOOTING` for each *identifier-2* having a control break (if any such report group is defined) will be presented.
- Next, the `CONTROL HEADING` for each *identifier-2* having a control break (if any such report group is defined) will be presented.
- The `CONTROL FOOTING` and `CONTROL HEADING` report groups will be presented in the sequence in which they are listed on the `CONTROL` clause.
- Only after this processing has occurred will the detail report group specified on the `GENERATE` be presented.

10. Each `RD` will have the following allocated for it:
    A. The `PAGE-COUNTER` special register (see [Special Registers], page 265), which will contain the current report page number.
        - This register will be set to a value of 1 when an `INITIATE` statement (see [INITIATE], page 343) is executed for the report and will be incremented by 1 each time the RWCS starts a new page of the report.
        - References to `PAGE-COUNTER` within the report section will be implicitly qualified with the name of the report to which the report group referencing the register belongs.
        - References to `PAGE-COUNTER` in the procedure division must be qualified with the appropriate report name if there are multiple `RD`s defined.
    B. The `LINE-COUNTER` special register, which will contain the current line number on the current page.
11. The `RD` must be followed by at least one 01-level report group definition.

6.6.1. Report Group Definitions

Report-Group-Definition Syntax

```
01 [ identifier-1 ]

[ LINE NUMBER IS { integer-1 [ [ ON NEXT PAGE ] ] } ]
  ~~~~           {                  ~~~~ ~~~~   }
                 { +|PLUS integer-1             }
                 {   ~~~~                       }
                 { ON NEXT PAGE                 }
                      ~~~~ ~~~~
[ NEXT GROUP IS { [ +|PLUS ] integer-2  } ]
  ~~~~ ~~~~~    {     ~~~~              }
                { NEXT|{NEXT PAGE}|PAGE }
                  ~~~~  ~~~~ ~~~~  ~~~~
[ TYPE IS { RH|{REPORT HEADING}                      } ]
  ~~~~    { ~~  ~~~~~~ ~~~~~~~                       }
          { PH|{PAGE HEADING}                        }
          { ~~  ~~~~ ~~~~~~~                         }
          { CH|{CONTROL HEADING} FINAL|identifier-2   }
          { ~~  ~~~~~~~ ~~~~~~~  ~~~~~               }
          { DE|DETAIL                                }
          { ~~ ~~~~~~                                }
          { CF|{CONTROL FOOTING} FINAL|identifier-2   }
          { ~~  ~~~~~~~ ~~~~~~~  ~~~~~               }
          { PF|{PAGE FOOTING}                        }
          {  ~~ ~~~~ ~~~~~~~                         }
          { RF|{REPORT FOOTING}                      }
            ~~  ~~~~~~ ~~~~~~~
.  [ REPORT-SECTION-Data-Item ]...
```

The syntax shown here documents how a report group is defined to a report. This syntax is valid only in the report section, and only then after an `RD`.

1. The reserved words `IS`, `NUMBER` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The `RH` and `REPORT HEADING` terms are interchangeable, as are `PH` and `PAGE HEADING`, `CH` and `CONTROL HEADING`, `DE` and `DETAIL`, `CF` and `CONTROL FOOTING`, `PF` and `PAGE FOOTING` as well as `RF` and `REPORT FOOTING`.
3. The report group being defined will be a part of the most-recently coded `RD`.
4. The `TYPE` (see [TYPE], page 222) clause specifies the type of report group being defined.
5. The level number used for a report group definition must be 01.

6. The optional *identifier-1* specification assigns a name to this report group so that the group may be referenced either by a `GENERATE` statement or on a `USE BEFORE REPORTING`.
7. No two report groups in the same report (`RD`) may named with the same *identifier-1*. There may, however, be multiple *identifier-1* definitions in different reports. In such instances, references to *identifier-1* must be qualified by the report name.
8. There may only be one report heading, report footing, final control heading, final control footing, page heading and page footing defined per report.
9. Report group declarations must be followed by at least one *`REPORT-SECTION-Data-Item`* with a level number in the range 02-49.
10. See [Data Description Clauses], page 154, for information on the usage of the various data description clauses.

6.6.2. REPORT SECTION Data Items

REPORT-SECTION-Data-Item Syntax

```
level-number [ identifier-1 ]

[ BLANK WHEN ZERO ]
  ~~~~~      ~~~~
[ COLUMN [ { NUMBER IS   } ] [ +|PLUS ] integer-1 ]
  ~~~      { ~~~~~~     }       ~~~~
           { NUMBERS ARE }
             ~~~~~~~
[ GROUP INDICATE ]
  ~~~~~ ~~~~~~~~
[ JUSTIFIED RIGHT ]
  ~~~~
[ LINE NUMBER IS { integer-2 [ [ ON NEXT PAGE ] } ]
  ~~~~           { +|PLUS integer-2 ~~~~ ~~~~   }
                 {   ~~~~                       }
                 { ON NEXT PAGE                 }
                      ~~~~ ~~~~
[ OCCURS [ integer-3 TO ] integer-4 TIMES
  ~~~~~~             ~~
    [ DEPENDING ON identifier-2 ]
      ~~~~~~~~~
    [ STEP integer-5 ]
      ~~~~
    [ VARYING identifier-3 FROM { identifier-4 } BY { identifier-5 } ]
      ~~~~~~~              ~~~~ { integer-6    } ~~ { integer-7    }
[ PICTURE IS picture-string ]
  ~~~
[ PRESENT WHEN condition-name ]
  ~~~~~~~ ~~~~
[ SIGN IS LEADING|TRAILING [ SEPARATE CHARACTER ] ]
  ~~~~    ~~~~~~~ ~~~~~~~~   ~~~~~~~~
[ { SOURCE IS literal-1|identifier-6 [ ROUNDED ]                    } ]
  { ~~~~~~                           ~~~~~~~                        }
  { SUM OF { identifier-7 }...  [ { RESET ON FINAL|identifier-8 } ] }
  { ~~~    { literal-2    }       { ~~~~~    ~~~~~              }   }
  { VALUE IS [ ALL ] literal-3    { UPON identifier-9           }   }
    ~~~~~      ~~~                  ~~~~
.  [ REPORT-SECTION-Data-Item ]...
```

Data item descriptions describing the report lines and fields that make up the substance of a report group immediately follow the definition of that group.

1. The reserved words `IS`, `NUMBER`, `OF`, `ON`, `RIGHT`, `TIMES` and `WHEN` (BLANK) are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved word `COLUMN` may be abbreviated as `COL`.
3. The reserved word `JUSTIFIED` may be abbreviated as `JUST`.
4. The reserved word `PICTURE` may be abbreviated as `PIC`.
5. The `SOURCE` (see [SOURCE], page 215), `SUM` (see [SUM], page 520) and `VALUE` (see [VALUE], page 244) clauses, valid only on an elementary item, are mutually-exclusive of each other.
6. Group items (those without `PICTURE` clauses) are frequently used to describe entire lines of a report, while elementary items (those with a picture clause) are frequently used to describe specific fields of information on the report. When this coding convention is being used, group items will have `LINE` (see [LINE], page 183) clauses and no `COLUMN` (see [COLUMN], page 166) clauses while elementary items will be specified the other way around.
7. See [Data Description Clauses], page 154, for information on the usage of the various data description clauses.

6.7. SCREEN SECTION

SCREEN-SECTION-Data-Item Syntax

```
level-number [ identifier-1 | FILLER ]
                              ~~~~~~

[ AUTO | AUTO-SKIP | AUTOTERMINATE ] [ BELL | BEEP ]
  ~~~~   ~~~~~~~~~   ~~~~~~~~~~~~~     ~~~~   ~~~~

[ BACKGROUND-COLOR|BACKGROUND-COLOUR IS integer-1 | identifier-2 ]
  ~~~~~~~~~~~~~~~ ~~~~~~~~~~~~~~~~~

[ BEFORE TIME | BASED ]
  ~~~~~~ ~~~~   ~~~~~

[ BLANK LINE|SCREEN ] [ ERASE EOL|EOS ]
  ~~~~~ ~~~~ ~~~~~~     ~~~~~ ~~~ ~~~

[ BLANK WHEN ZERO ] [ JUSTIFIED RIGHT ]
  ~~~~~      ~~~~     ~~~~

[ BLINK ] [ HIGHLIGHT | LOWLIGHT ] [ REVERSE-VIDEO ]
  ~~~~~     ~~~~~~~~~   ~~~~~~~~     ~~~~~~~~~~~~

[ COLUMN NUMBER IS [ +|PLUS ] integer-2 | identifier-3 ]
  ~~~                  ~~~~

[ CONSTANT | EMPTY CHECK ]
  ~~~~~~~    ~~~~~ ~~~~~

[ FOREGROUND-COLOR|FOREGROUND-COLOUR IS integer-3 | identifier-4 ]
  ~~~~~~~~~~~~~~~ ~~~~~~~~~~~~~~~~~

[ { FROM literal-1 | identifier-5 } ]
  { ~~~~                           }
  { TO identifier-5                }
  { ~~                             }
  { USING identifier-5             }
  { ~~~~~                          }
  { VALUE IS [ ALL ] literal-1     }
    ~~~~~      ~~~

[ FULL | LENGTH-CHECK ] [ REQUIRED | EMPTY-CHECK ] [ SECURE | NO-ECHO ]
  ~~~~   ~~~~~~~~~~~~     ~~~~~~~~   ~~~~~~~~~~~     ~~~~~~   ~~~~~~~

[ LEFTLINE ] [ OVERLINE ] [ UNDERLINE ]
  ~~~~~~~~     ~~~~~~~~     ~~~~~~~~~

[ LINE NUMBER IS [ +|PLUS ] integer-4 | identifier-6 ]
  ~~~~                ~~~~

[ LOWER | UPPER ]
  ~~~~~   ~~~~~

[ NO-ECHO | NO UPDATE ]
  ~~~~~~~   ~~ ~~~~~

[ OCCURS integer-5 TIMES ]
  ~~~~~~
```

```
[ PICTURE IS picture-string ]
  ~~~
[ PROMPT [ CHARACTER IS literal-2 | identifier-7 ]
  ~~~~~~   ~~~~~~~~~
[ SCROLL DOWN | SCROLL UP | SIZE | TIME OUT ]
  ~~~~~~ ~~~~   ~~~~~~ ~~   ~~~~   ~~~~ ~~~
[ SIGN IS LEADING|TRAILING [ SEPARATE CHARACTER ] ]
  ~~~~    ~~~~~~~ ~~~~~~~~   ~~~~~~~~
[ UPDATE ]
  ~~~~~~
.  [ SCREEN-SECTION-Data-Item ]...
```

The screen section describes the screens to be displayed during terminal/console I-O.

1. The reserved words `CHARACTER` (`SEPARATE` clause), `IS`, `NUMBER`, `RIGHT`, `TIMES` and `WHEN` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved word `COLUMN` may be abbreviated as `COL`.
3. The reserved word `PICTURE` may be abbreviated as `PIC`.
4. The following sets of reserved words are interchangeable:
   - `AUTO`, `AUTO-SKIP` and `AUTOTERMINATE`
   - `BACKGROUND-COLOR` and `BACKGROUND-COLOUR`
   - `BELL` and `BEEP`
   - `FOREGROUND-COLOR` and `FOREGROUND-COLOUR`
   - `FULL` and `LENGTH-CHECK`
   - `REQUIRED` and `EMPTY-CHECK`
   - `SECURE` and `NO-ECHO`
5. Data items defined in the screen section describe input, output or combination screen layouts to be used with `ACCEPT data-item` statement (see [ACCEPT data-item], page 272) or `DISPLAY data-item` statement (see [DISPLAY data-item], page 311) statements. These screen layouts may define the entire available screen area or any subset of it.
6. The term *available screen area* is a nebulous one in those environments where command-line shell sessions are invoked within a graphical user-interface environment, as will be the case on Windows, OSX and most Unix/Linux systems — these environments allow command-line session windows to exist with a variable number of available screen rows and columns. When you are designing GnuCOBOL screens, you need to do so with an awareness of the logical screen row/column geometry the program will be executing within.
7. Data items with level numbers 01 (Constants), 66, 78 and 88 may be used in the screen section; they have the same syntax, rules and usage as they do in the other data division sections.

8. Without `LINE` (see [LINE], page 183) or `COLUMN` (see [COLUMN], page 166) clauses, screen section fields will display on the console window beginning at whatever line/column coordinate is stated or implied by the `ACCEPT data-item` or `DISPLAY data-item` statement that presents the screen item. After a field is presented to the console window, the next field will be presented immediately following that field.
9. A `LINE` clause explicitly stated in the definition of a screen section data item will override any `LINE` clause included on the `ACCEPT data-item` or `DISPLAY data-item` statement that presents that data item to the screen. The same is true of `COLUMN` clauses.
10. The `Tab` and `Back-Tab` (`Shift-Tab` on most keyboards) keys will position the cursor from field to field in the line/column sequence in which the fields occur on the screen at execution time, regardless of the sequence in which they were defined in the screen section.
11. See [Data Description Clauses], page 154, for information on the usage of the various data description clauses.

6.8. Special Data Items

6.8.1. 01-Level Constants

01-Level-Constant Syntax

```
01 constant-name-1 CONSTANT [ IS GLOBAL ]
                   ~~~~~~~~      ~~~~~~

  { AS { arithmetic-espression-1             } } .
  {    { literal-1                           } }
  {    { { BYTE-LENGTH } OF { identifier-1 } } }
  {    { { ~~~~~~~~~~~ }    { usage-name   } } }
  {    { { LENGTH      }                     } }
  {        ~~~~~~                            }
  { FROM CDF-variable-name-1                 }
    ~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`, `SCREEN`.

The 01-level constant is one of five types of compilation-time constants that can be declared within a program. The other four types are `>>DEFINE` CDF directive (see [>>DEFINE], page 68) constants, `>>SET` CDF directive (see [>>SET], page 71) constants, 78-level constants (see [78-Level Data Items], page 152, and arithmetic-espression-1).

1. The reserved words `AS`, `IS` and `OF` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. See [GLOBAL], page 176, for information on the `GLOBAL` option.
3. This particular type of constant declaration provides the ability to determine the length of a data item or the storage size associated with a particular numeric `USAGE` (see [USAGE], page 232) type — something not possible with the other types of constants.
4. Constants defined in this way become undefined once an `END PROGRAM` or `END FUNCTION` is encountered in the input source.
5. Data descriptions of this form do not actually allocate any storage — they merely define a name (*constant-name-1*) that may be used anywhere a numeric literal (see `BYTE-LENGTH` or `LENGTH` options) or a literal of the same type as *literal-1* may be used.
6. The *constant-name-1* name may not be referenced on a CDF directive.
7. Care must be taken that *constant-name-1* does not duplicate any other data item name that has been defined in the program as references to that data item name will refer to the constant and not the data item. The GnuCOBOL compiler will not issue a warning about this condition.
8. The value specified for *usage-name* may be any `USAGE` that does not use a `PICTURE` (see [PICTURE], page 194) clause. These would be any of `BINARY-C-LONG`, `BINARY-CHAR`, `BINARY-DOUBLE`, `BINARY-LONG`, `BINARY-SHORT`, `COMP-1` (or `COMPUTATIONAL-1`),

COMP-2 (or COMPUTATIONAL-2), FLOAT-DECIMAL-16, FLOAT-DECIMAL-34, FLOAT-LONG, FLOAT-SHORT, POINTER, or PROGRAM-POINTER.

9. The BYTE-LENGTH clause will produce a numeric value for *constant-name-1* identical to that which would be returned by the BYTE-LENGTH intrinsic function executed against *identifier-1* or a data item declared with a USAGE of *usage-name*.
10. The LENGTH clause will produce a numeric value for *constant-name-1* identical to that which would be returned by the LENGTH intrinsic function executed against *identifier-1* or a data item declared with a USAGE of *usage-name*.

Here is an example of using the option arithmetic-espression.

```
PROGRAM-ID. TESTCONST.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 A   CONSTANT 2.
01 B   CONSTANT ( (3 + 2) * A).
01 CON CONSTANT (A ** B).
01 MYDATA PIC X(CON).
PROCEDURE DIVISION.
 DISPLAY CON
 ACCEPT omitted
 MOVE "12345678901234567890123456789O" TO MYDATA
  DISPLAY MYDATA
 ACCEPT omitted
 GOBACK.
```

Here is the listing of a GnuCOBOL program that uses 01-level constants to display the length (in bytes) of the various picture-less usage types.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Usage-Lengths.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Len-BINARY-C-LONG    CONSTANT AS LENGTH OF BINARY-C-LONG.
01  Len-BINARY-CHAR      CONSTANT AS LENGTH OF BINARY-CHAR.
01  Len-BINARY-DOUBLE    CONSTANT AS LENGTH OF BINARY-DOUBLE.
01  Len-BINARY-LONG      CONSTANT AS LENGTH OF BINARY-LONG.
01  Len-BINARY-SHORT     CONSTANT AS LENGTH OF BINARY-SHORT.
01  Len-COMP-1           CONSTANT AS LENGTH OF COMP-1.
01  Len-COMP-2           CONSTANT AS LENGTH OF COMP-2.
01  Len-FLOAT-DECIMAL-16 CONSTANT AS LENGTH OF FLOAT-DECIMAL-16.
01  Len-FLOAT-DECIMAL-34 CONSTANT AS LENGTH OF FLOAT-DECIMAL-34.
01  Len-FLOAT-LONG       CONSTANT AS LENGTH OF FLOAT-LONG.
01  Len-FLOAT-SHORT      CONSTANT AS LENGTH OF FLOAT-SHORT.
01  Len-POINTER          CONSTANT AS LENGTH OF POINTER.
01  Len-PROGRAM-POINTER  CONSTANT AS LENGTH OF PROGRAM-POINTER.
PROCEDURE DIVISION.
```

```
000-Main.
    DISPLAY "On this system, with this build of GnuCOBOL, the"
    DISPLAY "PICTURE-less USAGE's have these lengths (in bytes):"
    DISPLAY " "
    DISPLAY "BINARY-C-LONG:    " Len-BINARY-C-LONG
    DISPLAY "BINARY-CHAR:      " Len-BINARY-CHAR
    DISPLAY "BINARY-DOUBLE:    " Len-BINARY-DOUBLE
    DISPLAY "BINARY-LONG:      " Len-BINARY-LONG
    DISPLAY "BINARY-SHORT:     " Len-BINARY-SHORT
    DISPLAY "COMP-1:           " Len-COMP-1
    DISPLAY "COMP-2:           " Len-COMP-2
    DISPLAY "FLOAT-DECIMAL-16: " Len-FLOAT-DECIMAL-16
    DISPLAY "FLOAT-DECIMAL-34: " Len-FLOAT-DECIMAL-34
    DISPLAY "FLOAT-LONG:       " Len-FLOAT-LONG
    DISPLAY "FLOAT-SHORT:      " Len-FLOAT-SHORT
    DISPLAY "POINTER:          " Len-POINTER
    DISPLAY "PROGRAM-POINTER:  " Len-PROGRAM-POINTER
    STOP RUN.
```

The output of this program, on a Windows 7 system with a 32-bit MinGW build of GnuCOBOL is:

```
On this system, with this build of GnuCOBOL, the
PICTURE-less USAGE's have these lengths (in bytes):

BINARY-C-LONG:    4
BINARY-CHAR:      1
BINARY-DOUBLE:    8
BINARY-LONG:      4
BINARY-SHORT:     2
COMP-1:           4
COMP-2:           8
FLOAT-DECIMAL-16: 8
FLOAT-DECIMAL-34: 16
FLOAT-LONG:       8
FLOAT-SHORT:      4
POINTER:          4
PROGRAM-POINTER:  4
```

```
The output of this program, on a Linux X64 system running cobc (GnuCOBOL) 3.1.2.0 is:

On this system, with this build of GnuCOBOL, the
PICTURE-less USAGE's have these lengths (in bytes):

BINARY-C-LONG:    8
BINARY-CHAR:      1
BINARY-DOUBLE:    8
```

```
BINARY-LONG:      4
BINARY-SHORT:     2
COMP-1:           4
COMP-2:           8
FLOAT-DECIMAL-16: 8
FLOAT-DECIMAL-34: 16
FLOAT-LONG:       8
FLOAT-SHORT:      4
POINTER:          8
PROGRAM-POINTER:  8

Spot the differences between 32 and 64 bit.
```

6.8.2. 66-Level Data Items

66-Level-Data-Item Syntax

```
66 identifier-1 RENAMES identifier-2 [ THRU|THROUGH identifier-3 ] .
                ~~~~~~~                ~~~~ ~~~~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`

A 66-level data item regroups previously defined items by specifying alternative, possibly overlapping, groupings of elementary data items.

1. The reserved words `THRU` and `THROUGH` are interchangeable.
2. A level-66 data item cannot rename a level-66, level-01, level-77, or level-88 data item.
3. There may be multiple level-66 data items that rename data items contained within the same 01-level record description.
4. All `RENAMES` entries associated with one logical record must immediately follow that record's last data description entry.

6.8.3. 77-Level Data Items

77-Level-Data-Item Syntax

```
77 identifier-1 [ IS GLOBAL|EXTERNAL ]
                  ~~~~~~ ~~~~~~~~

[ BASED ]
  ~~~~~

[ BLANK WHEN ZERO ]
  ~~~~~      ~~~~

[ JUSTIFIED RIGHT ]
  ~~~~

[ PICTURE IS picture-string ]
  ~~~

[ REDEFINES identifier-5 ]
  ~~~~~~~~~

[ SIGN IS LEADING|TRAILING [ SEPARATE CHARACTER ] ]
  ~~~~    ~~~~~~~ ~~~~~~~~   ~~~~~~~~

[ SYNCRONIZED|SYNCHRONISED [ LEFT|RIGHT ] ]
  ~~~~         ~~~~           ~~~~ ~~~~~

[ USAGE IS data-item-usage ]
  ~~~~~

[ VALUE IS [ ALL ] literal-1 ] .
  ~~~~~      ~~~
```

The `LEFT` and `RIGHT` (SYNCRONIZED) clauses are syntactically recognized but are otherwise non-functional.

This syntax is valid in the following sections: `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`

The intent of a 77-level item is to be able to create a stand-alone elementary data item.

1. The reserved words `CHARACTER`, `IS`, `RIGHT` (JUSTIFIED) and `WHEN` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved word `JUSTIFIED` may be abbreviated as `JUST`, the reserved word `PICTURE` may be abbreviated as `PIC` and the reserved words `SYNCRONIZED` and `SYNCHRONISED` may be abbreviated as `SYNC`.
3. New programs requiring a stand-alone elementary item should be coded to use a level number of 01 rather than 77.
4. See [Data Description Clauses], page 154, for information on the usage of the various data description clauses.

6.8.4. 78-Level Data Items

78-Level-Constant Syntax

```
78 constant-name-1 VALUE IS literal-1 .
                   ~~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`, `SCREEN`

The 78-level constant is one of four types of compilation-time constants that can be declared within a program. The other three types are `>>DEFINE` CDF directive (see [>>DEFINE], page 68) constants, `>>SET` CDF directive (see [>>SET], page 71) constants and 01-level constants (see [01-Level Constants], page 146).

1. The reserved word `IS` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. Constants defined in this way become undefined once an `END PROGRAM` or `END FUNCTION` is encountered in the input source.
3. Data descriptions of this form do not actually allocate any storage — they merely define a name (*constant-name-1*) that may be used anywhere a literal of the same type as *literal-1* may be used.
4. The *constant-name-1* name may not be referenced on a CDF directive.
5. Care must be taken that *constant-name-1* does not duplicate any other data item name that has been defined in the program as references to that data item name will refer to the constant and not the data item. The GnuCOBOL compiler will not issue a warning about this condition.

6.8.5. 88-Level Data Items

88-Level-Data-Item Syntax

```
88 condition-name-1 { VALUE IS   } {literal-1 [ THRU|THROUGH literal-2 ]}...
                    { ~~~~~      }              ~~~~ ~~~~~~~
                    { VALUES ARE }
                      ~~~~~~

  [ WHEN SET TO FALSE IS literal-3 ] .
                ~~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`, `REPORT`, `SCREEN`

Condition names are Boolean (*i.e.* `TRUE` / `FALSE`) data items that receive their `TRUE` and `FALSE` values based upon the values of the non 88-level data item whose definition they immediately follow.

1. The reserved words `ARE`, `IS`, `SET` and `TO` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved words `THRU` and `THROUGH` are interchangeable.
3. Condition names are always defined subordinate to another (non 88-level) data item. That data item must be an elementary item. Whenever the parent data item assumes one of the values specified on the 88-level item's `VALUE` (see [VALUE], page 244) clause, *condition-name-1* will take on the value of `TRUE`.
4. Condition names do not occupy any storage.
5. The optional `THROUGH` clause allows a range of possible `TRUE` values to be specified.
6. Whenever the parent data item assumes any value *except* one of the values specified on *condition-name-1*'s `VALUE` clause, *condition-name-1* will take on the value of FALSE.
7. Executing the statement `SET` *`condition-name-1`* `TO TRUE` will cause *condition-name-1*'s parent data item to take on the first value specified on *condition-name-1*'s `VALUE` clause.
8. Executing the statement `SET` *`condition-name-1`* `TO FALSE` will cause *condition-name-1*'s parent data item to take on the value specified on *condition-name-1*'s `FALSE` clause. If *condition-name-1* does not have a `FALSE` clause, the `SET` (see [SET], page 382) statement will generate an error message at compilation time.
9. See [Condition Names], page 45, for more information.

6.9. Data Description Clauses

6.9.1. ANY LENGTH

ANY LENGTH Attribute Syntax

```
ANY LENGTH
~~~ ~~~~~~
```

This syntax is valid in the following sections: `LINKAGE`

Data items declared with the `ANY LENGTH` attribute have no fixed compile-time length. Such items may only be defined in the linkage section of a subprogram as they may only serve as subroutine argument descriptions. These items must have a `PICTURE` (see [PICTURE], page 194) clause that specifies exactly one A, X or 9 symbol.

1. The `ANY LENGTH` and `BASED` (see [BASED], page 160) clauses cannot be used together in the same data item description.

6.9.2. AUTO

AUTO Attribute Syntax

```
AUTO
~~~~
```

This syntax is valid in the following sections: `SCREEN`

A field whose description includes this attribute will cause the cursor to automatically advance to the next input-enabled field of a screen if the field is completely filled with input data.

1. The `AUTO`, `AUTO-SKIP` (see [AUTO-SKIP], page 156) and `AUTOTERMINATE` (see [AUTOTERMINATE], page 157) clauses are interchangeable, and may not be used together in the same data item description.

6.9.3. AUTO-SKIP

AUTO-SKIP Attribute Syntax

```
AUTO-SKIP
~~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

A field whose description includes this attribute will cause the cursor to automatically advance to the next input-enabled field of a screen if the field is completely filled with input data.

1. The `AUTO` (see [AUTO], page 155), `AUTO-SKIP` and `AUTOTERMINATE` (see [AUTOTERMINATE], page 157) clauses are interchangeable, and may not be used together in the same data item description.

6.9.4. AUTOTERMINATE

AUTOTERMINATE Attribute Syntax

```
AUTOTERMINATE
~~~~~~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

A field whose description includes this attribute will cause the cursor to automatically advance to the next input-enabled field of a screen if the field is completely filled with input data.

1. The `AUTO` (see [AUTO], page 155), `AUTO-SKIP` (see [AUTO-SKIP], page 156) and `AUTOTERMINATE` clauses are interchangeable, and may not be used together in the same data item description.

6.9.5. BACKGROUND-COLOR

BACKGROUND-COLOR Attribute Syntax

```
BACKGROUND-COLOR|BACKGROUND-COLOUR IS integer-1 | identifier-1
~~~~~~~~~~~~~~~ ~~~~~~~~~~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause is used to specify the screen background color of the screen data item or the default screen background color of subordinate items if used on a group item.

1. The reserved word `IS` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. The reserved words `BACKGROUND-COLOR` and `BACKGROUND-COLOUR` are interchangeable.
3. You specify colors by number (0-7), or by using the constant names provided in the `screenio.cpy` copybook (provided with all GnuCOBOL source distributions).
4. Colors may also be specified using a numeric non-edited identifier whose value is in the range 0-7.

For composite `DISPLAY`'s, the attributes are always only applied to the previous source-item but the following also allows a change by variable or literal *i.e.*

```
DISPLAY "Name: "   BACKGROUND-COLOR COB-YELLOW
         NAME-VAR  BACKGROUND-COLOR COB-BLACK
END-DISPLAY
```

See [Color Palette and Video Attributes], page 23, for more information on screen colors and video attributes.

6.9.5B. BEFORE TIME

BEFORE TIME Attribute Syntax

```
BEFORE TIME
~~~~~~ ~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause is used to specify time sequence etc, etc, etc. NEEDS CONTENT HERE.

6.9.6. BASED

BASED Attribute Syntax

```
BASED
~~~~~
```

This syntax is valid in the following sections: `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`

Data items declared with `BASED` are allocated no storage at compilation time. At run-time, the `ALLOCATE` (see [ALLOCATE], page 290) or `SET ADDRESS` (see [SET ADDRESS], page 384) statements are used to allocate space for and (optionally) initialize such items.

1. The `BASED` and `ANY LENGTH` (see [ANY LENGTH], page 154) clauses cannot be used together in the same data item description.
2. The `BASED` clause may only be used on level 01 and level 77 data items.

6.9.7. BEEP

BEEP Attribute Syntax

```
BEEP
~~~~
```

This syntax is valid in the following sections: `SCREEN`

1. The `BEEP` and `BELL` (see [BELL], page 162) clauses are interchangeable, and may not be used together in the same data item description.
2. Use this clause to cause an audible tone to occur when the screen item is `DISPLAY`ed.

6.9.8. BELL

BELL Attribute Syntax

```
BELL
~~~~
```

This syntax is valid in the following sections: `SCREEN`

1. The `BEEP` (see [BEEP], page 161) and `BELL` clauses are interchangeable, and may not be used together in the same data item description.
2. Use this clause to cause an audible tone to occur when the screen item is `DISPLAY`ed.

6.9.9. BLANK

BLANK Attribute Syntax

```
BLANK LINE|SCREEN
~~~~~ ~~~~ ~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause will blank out either the entire screen (`BLANK SCREEN`) or just the line upon which data is about to be displayed (`BLANK LINE`).

1. Blanked-out areas will have their foreground and background colors set to the attributes of the field containing the `BLANK` clause.
2. This clause is useful when one screen section item is being displayed over the top of a previously-displayed one.

6.9.10. BLANK WHEN ZERO

BLANK-WHEN-ZERO Attribute Syntax

```
BLANK WHEN ZERO
~~~~~      ~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`, `REPORT`, `SCREEN`

This clause will cause that item's value to be automatically transformed into spaces if a value of 0 is ever `MOVE`d to the item.

1. The reserved word `WHEN` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. This clause may only be used on a `PIC 9` data item with a `USAGE` (see [USAGE], page 232) of `DISPLAY`.

6.9.11. BLINK

BLINK Attribute Syntax

```
BLINK
~~~~~
```

This syntax is valid in the following sections: `SCREEN`

The `BLINK` clause modifies the visual appearance of the displayed field by making the field contents blink.

See [Color Palette and Video Attributes], page 23, for more information on screen colors and video attributes.

6.9.12. COLUMN

COLUMN (REPORT SECTION) Clause Syntax

```
COLUMN [ { NUMBER IS   } ] [ +|PLUS ] integer-1 ]
~~~      { NUMBERS ARE }       ~~~~
```

COLUMN (SCREEN SECTION) Clause Syntax

```
COLUMN NUMBER IS [ +|PLUS ] integer-2 | identifier-3 ]
~~~                 ~~~~
```

This syntax is valid in the following sections: `REPORT`, `SCREEN`

The `COLUMN` clause provides the means of stating in which column a field should be presented on the console window (screen section) or a report (report section).

1. The reserved words `ARE`, `IS`, `NUMBER` and `NUMBERS` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved word `COLUMN` may be abbreviated as `COL`.
3. The line location of a report section or screen section field will be determined by the `LINE` (see [LINE], page 183) clause.
4. The value of *integer-1* must be 1 or greater.
5. If *identifier-1* is used to specify either an absolute or relative column position, *identifier-1* must be defined as a numeric item of any `USAGE` (see [USAGE], page 232) other than `COMPUTATIONAL-1` or `COMPUTATIONAL-2`, without editing symbols. The value of *identifier-1* at the time the screen data item is presented must be 1 or greater. Note that a `COMPUTATIONAL-1` or `COMPUTATIONAL-2` identifier will be accepted by the compiler, but will produce unpredictable results at run-time.
6. The column coordinate of a field may be stated on an absolute basis (*i.e.* `COLUMN 5`) or on a relative basis based upon the end of the previously-presented field (*i.e.* `COLUMN PLUS 1`).
7. The symbol '`+`' may be used in lieu of the word `PLUS`, if desired; if symbol '`+`' is used, however, there must be at least one space separating it from *integer-1*. Failure to include this space will cause the symbol '`+`' sign to be simply treated as part of *integer-1* and will treat the `COLUMN` clause as an absolute column specification rather than a relative one.
8. Using relative column positioning (`COLUMN PLUS`) has slightly different behaviour depending upon the section in which the clause is used, as follows:
   A. When used on a report section data item, `COLUMN PLUS` will position the start of the new field's value such that there are *integer-1* blank columns between the end of the previous field and the beginning of this field.

If a report data item's description includes the `SOURCE` (see [SOURCE], page 215), `SUM` (see [SUM], page 520) or `VALUE` (see [VALUE], page 244) clause but has no `COLUMN` clause, `COLUMN PLUS 1` will be assumed.

B. When used on a screen section data item, `COLUMN PLUS` will position the new field so that it begins exactly *integer-1* or *identifier-1* characters past the *last* character of the previous field. Thus, `COLUMN PLUS 1` will leave no blank positions between the end of the previous field and the start of this one.

   If a screen data item's description includes the `FROM` (see [FROM], page 174), `TO` (see [TO], page 221), `USING` (see [USING], page 243) or `VALUE` (see [VALUE], page 244) clause but has no `COLUMN` clause, the new screen field will begin at the column coordinate of the last character of the previous field.

6.9.13. CONSTANT

CONSTANT Attribute Syntax

```
CONSTANT
~~~~~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`, `SCREEN`

This option signifies that the 01-level data item in whose declaration `CONSTANT` is specified will be treated as a symbolic name for a literal value, usable wherever a literal of the appropriate type could be used.

1. The value of a data item defined as a constant cannot be changed at run-time. In fact, it is not syntactically acceptable to use such a data item as the destination field of any procedure division statement that stores a value.
2. See [01-Level Constants], page 146, for additional information.

6.9.14. EMPTY-CHECK

EMPTY-CHECK Attribute Syntax

```
EMPTY-CHECK
~~~~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause forces the user to enter data into the field it is specified on (or into all subordinate input-capable fields if `EMPTY-CHECK` is specified on a group item).

1. The `EMPTY-CHECK` and `REQUIRED` (see [REQUIRED], page 206) clauses are interchangeable, and may not be used together in the same data item description.
2. In order to take effect, the user must first move the cursor into the field having this clause in its definition.
3. The `ACCEPT data-item` statement (see [ACCEPT data-item], page 272) will ignore the Enter key and any other cursor-moving keystrokes that would cause the cursor to move to another screen item *unless* data has been entered into the field. Function keys will still be allowed to terminate the `ACCEPT`.
4. In order to be functional, this attribute must be supported by the underlying "curses" package your GnuCOBOL implementation was built with. As of this time, the "PDCurses" package (used for native Windows or MinGW builds) does not support `EMPTY-CHECK`.

6.9.15. ERASE

ERASE Clause Syntax

```
ERASE EOL|EOS
~~~~~ ~~~ ~~~
```

This syntax is valid in the following sections: `SCREEN`

`ERASE` will blank-out screen contents from the location where the screen data item whose description contains this clause will be displayed, forward until the end of the screen (`ERASE EOS`)

1. Erased areas will have their foreground and background colors set to the attributes of the field containing the `ERASE` clause.
2. This clause is useful when one screen section item is being displayed over the top of a previously-displayed one.

See [Color Palette and Video Attributes], page 23, for more information on screen colors and video attributes.

6.9.16. EXTERNAL

EXTERNAL Attribute Syntax

```
EXTERNAL
~~~~~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`

This clause marks a data item description, `FD` or `SD` see [File/Sort-Description], page 124, as being shareable with other programs executed from the same execution thread.

1. By specifying the `EXTERNAL` clause on either an `FD` or an `SD`, the file description is capable of being shared between all programs executed from the same execution thread, provided an `EXTERNAL` clause is coded with the file's description in *each* program requiring it. This sharing allows the file to be opened, read and/or written and closed in different programs. This sharing applies to the record descriptions subordinate to the file description too.
2. By specifying the `EXTERNAL` clause on the description of a data item, the data item is capable of being shared between all programs executed from the same execution thread, provided the data item is coded (with an `EXTERNAL` clause) in each program requiring it.
3. The following points apply to the specification of `EXTERNAL` in a data item's definition:
   A. The `EXTERNAL` clause may only be specified at the 77 or 01 level.
   B. An `EXTERNAL` item must have a data name and that name cannot be `FILLER`.
   C. `EXTERNAL` cannot be combined with `BASED` (see [BASED], page 160), `GLOBAL` (see [GLOBAL], page 176) or `REDEFINES` (see [REDEFINES], page 204).

6.9.17. FALSE

FALSE Clause Syntax

```
WHEN SET TO FALSE IS literal-1
            ~~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`, `REPORT`, `SCREEN`

This clause, which may only appear on the definition of a level-88 condition name, is used to specify the value of the data item that serves as the parent of the level-88 condition name that will force the condition name to assume a value of `FALSE`.

1. The reserved words `IS`, `SET`, `TO` and `WHEN` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. See [88-Level Data Items], page 153, or See [Condition Names], page 45, for more information.

6.9.18. FOREGROUND-COLOR

FOREGROUND-COLOR Attribute Syntax

```
FOREGROUND-COLOR|FOREGROUND-COLOUR IS integer-1 | identifier-1
~~~~~~~~~~~~~~~~ ~~~~~~~~~~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause is used to specify the color of text within a screen data item or the default text color of subordinate items if used on a group item.

1. The reserved word `IS` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. The reserved words `FOREGROUND-COLOR` and `FOREGROUND-COLOUR` are interchangeable.
3. You specify colors by number (0-7), or by using the constant names provided in the `screenio.cpy` copybook (which is provided with all GnuCOBOL source distributions).
4. Colors may also be specified using a numeric non-edited identifier whose value is in the range 0-7.

See [Color Palette and Video Attributes], page 23, for more information on screen colors and video attributes.

6.9.19. FROM

FROM Clause Syntax

```
FROM literal-1 | identifier-5
~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause is used to specify either the data item a screen section field is to obtain its value from when the screen is displayed, or a literal that will specify the value of that same field.

1. The `FROM`, `TO` (see [TO], page 221), `USING` (see [USING], page 243) and `VALUE` (see [VALUE], page 244) clauses are mutually-exclusive in any screen section data item's definition.

6.9.20. FULL

FULL Attribute Syntax

```
FULL
~~~~
```

This syntax is valid in the following sections: `SCREEN`

The `FULL` clause forces the user to enter data into the field it is specified on (or into all subordinate input-capable fields if specified on a group item) sufficient to fill every character position of the field.

1. The `FULL` and `LENGTH-CHECK` (see [LENGTH-CHECK], page 182) clauses are interchangeable, and may not be used together in the same data item description.
2. In order for this clause to take effect at execution time, the user must move the cursor into the field having this clause in its definition.
3. The `ACCEPT data-item` statement (see [ACCEPT data-item], page 272) will ignore the Enter key and any other cursor-moving keystrokes that would cause the cursor to move to another screen item unless the proper amount of data has been entered into the field. Function keys will still be allowed to terminate the `ACCEPT`, however.
4. In order to be functional, this attribute must be supported by the underlying "curses" package your GnuCOBOL implementation was built with. As of this time, the "PDCurses" package (used for native Windows or MinGW builds) does not support `FULL`.

6.9.21. GLOBAL

GLOBAL Attribute Syntax

```
GLOBAL
~~~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `REPORT`

This clause marks a data item, 01-level constant, `FD` (see [File/Sort-Description], page 124), `SD` (see [File/Sort-Description], page 124) or an `RD` (see [REPORT SECTION], page 135) as being shareable with any nested subprograms.

1. By specifying the `GLOBAL` clause on the description of a file or a report, that description is capable of being shared between a program and any nested subprograms within it, provided the `FD`, `SD` or `RD` is coded (with a `GLOBAL` clause) in each nested subprogram requiring it. This sharing allows the file to be opened, read and/or written and closed or the report to be initiated or terminated in those programs. Separately compiled programs may not share a `GLOBAL` file description, but they may share an `EXTERNAL` (see [EXTERNAL], page 171) file description. This sharing applies to the record descriptions subordinate to the file description and the report groups subordinate to the `RD` also.
2. By specifying the `GLOBAL` clause on the description of a data item, the data item is capable of being shared between a program and any nested subprograms within it, provided the data item is coded (with a `GLOBAL` clause) in each program requiring it.
3. The following points apply to the specification of `GLOBAL` in a data item's definition:
   A. The `GLOBAL` clause may only be specified at the 77 or 01 level.
   B. A `GLOBAL` item must have a data name and that name cannot be `FILLER`.
   C. `GLOBAL` cannot be combined with `EXTERNAL` (see [EXTERNAL], page 171), `REDEFINES` (see [REDEFINES], page 204) or `BASED` (see [BASED], page 160).

6.9.22. GROUP INDICATE

GROUP-INDICATE Attribute Syntax

```
GROUP INDICATE
~~~~~ ~~~~~~~~
```

This syntax is valid in the following sections: `REPORT`

The `GROUP INDICATE` clause specifies that the data item in whose definition the clause appears will be presented only in very limited circumstances.

1. This clause may only appear within a `DETAIL` report group (see [TYPE], page 222).
2. When this clause is present, the data item in question will be presented only under the following circumstances:
   A. On the first presentation of the detail group following the `INITIATE` (see [INITIATE], page 343) of the report.
   B. On the first presentation of the detail group after every new page is started.
   C. On the first presentation of the detail group after any control break occurs.

6.9.23. HIGHLIGHT

HIGHLIGHT Attribute Syntax

```
HIGHLIGHT
~~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause controls the intensity of text (`FOREGROUND-COLOR` (see [FOREGROUND-COLOR], page 173)) by setting that intensity to its highest of three possible settings.

1. This clause, along with `LOWLIGHT` (see [LOWLIGHT], page 186), are intended to provide a three-level intensity scheme (`LOWLIGHT` ... nothing (Normal) ... `HIGHLIGHT`).

See [Color Palette and Video Attributes], page 23, for more information on screen colors and video attributes.

6.9.24. JUSTIFIED

JUSTIFIED Attribute Syntax

```
JUSTIFIED RIGHT
~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`, `REPORT`, `SCREEN`

The presence of a `JUSTIFIED RIGHT` clause in a data item's definition alters the manner in which data is stored into the field from the default 'left-justified, space filled' behaviour to 'right justified, space filled'.

1. The reserved word `RIGHT` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. The reserved word `JUSTIFIED` may be abbreviated as `JUST`.
3. This clause is valid only on alphabetic (`PIC A`) or alphanumeric (`PIC X`) data items.
4. The presence or absence of this clause influences the behaviour of the `MOVE` (see [MOVE], page 352) statement as well as the `FROM` (see [FROM], page 174), `SOURCE` (see [SOURCE], page 215) and `USING` (see [USING], page 243) data item description clauses.
5. If the value being stored into the field is the same length as the receiving field, the presence or absence of the `JUSTIFIED RIGHT` clause on that field's description is irrelevant.
6. The following examples illustrate the behaviour of the presence and absence of the `JUSTIFIED RIGHT` clause when the field size is different than that of the value being stored. In these examples, the symbol *b* represents a space.

   When the value is *shorter* than the field size:

   Without `JUSTIFIED`

   ```
   01  A PIC X(6).
   MOVE @code{ABC} TO A
   ```

   Result

   ABC*bbb*

   With `JUSTIFIED`

   ```
   01  A PIC X(6) JUSTIFIED RIGHT.
   MOVE @code{ABC} TO A
   ```

   Result

   *bbb*ABC

When the value is *longer* than the field size:

Without JUSTIFIED

```
01  A PIC X(6).
MOVE 'ABCDEFGHI' TO A
```

Result

```
ABCDEF
```

With JUSTIFIED

```
01  A PIC X(6) JUSTIFIED RIGHT.
MOVE 'ABCDEFGHI' TO A
```

Result

```
DEFGHI
```

6.9.25. LEFTLINE

LEFTLINE Attribute Syntax

```
LEFTLINE
~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

The `LEFTLINE` clause will introduce a vertical line at the left edge of a screen field.

1. The `LEFTLINE`, `OVERLINE` (see [OVERLINE], page 193) and `UNDERLINE` (see [UNDERLINE], page 229) clauses may be used in any combination in a single field's description.
2. This clause is essentially non-functional when used within Windows command shell (cmd.exe) environments and running programs compiled using a GnuCOBOL implementation built using "PDCurses" (such as Windows/MinGW builds).
3. Whether or not this clause operates on Cygwin or UNIX/Linux/OSX systems will depend upon the video attribute capabilities of the terminal output drivers and "curses" software being used.

See [Color Palette and Video Attributes], page 23, for more information on screen colors and video attributes.

6.9.26. LENGTH-CHECK

LENGTH-CHECK Attribute Syntax

```
LENGTH-CHECK
~~~~~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

The `LENGTH-CHECK` clause forces the user to enter data into the field it is specified on (or into all subordinate input-capable fields if specified on a group item) sufficient to fill every character position of the field.

1. The `FULL` (see [FULL], page 175) and `LENGTH-CHECK` clauses are interchangeable, and may not be used together in the same data item description.
2. In order for this clause to take effect at execution time, the user must move the cursor into the field having this clause in its definition.
3. The `ACCEPT data-item` statement (see [ACCEPT data-item], page 272) will ignore the Enter key and any other cursor-moving keystrokes that would cause the cursor to move to another screen item unless the proper amount of data has been entered into the field. Function keys will still be allowed to terminate the `ACCEPT`, however.
4. In order to be functional, this attribute must be supported by the underlying "curses" package your GnuCOBOL implementation was built with. As of this time, the "PDCurses" package (used for native Windows or MinGW builds) does not support `LENGTH-CHECK`.

6.9.27. LINE

LINE (REPORT SECTION) Clause Syntax

```
LINE NUMBER IS { integer-2 [ [ ON NEXT PAGE ] }
~~~~           {                  ~~~~ ~~~~   }
               { +|PLUS integer-2             }
               {   ~~~~                       }
               { ON NEXT PAGE                 }
                    ~~~~ ~~~~
```

LINE (SCREEN SECTION) Clause Syntax

```
[ LINE NUMBER IS [ +|PLUS ] integer-4 | identifier-6 ]
  ~~~~                ~~~~
```

This syntax is valid in the following sections: `REPORT`, `SCREEN`

This clause provides a means of explicitly stating on which line a field should be presented on the console window (screen section) or on a report (report section).

1. The reserved words `IS`, `NUMBER` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The following points document the use of format 1 of the `LINE` clause:
   A. The column location of a report item will be determined by the `COLUMN` (see [COLUMN], page 166) clause.
   B. The value of *integer-1* must be 1 or greater.
   C. The report line number upon which the data item containing this clause along with any subordinate data items will be presented may be stated on an absolute basis (*i.e.* `LINE 5`) or on a relative basis based upon the previously-displayed line (*i.e.* `LINE PLUS 1`).
   D. The symbol '`+`' may be used in lieu of the word `PLUS`, if desired; if '`+`' is used, however, there must be at least one space separating it from *integer-1*. Failure to include this space will cause the '`+`' to be simply treated as part of *integer-1* and will treat the LINE clause as an absolute line specification rather than a relative one.
   E. The optional `NEXT PAGE` clause specifies that — regardless of whether or not the report group containing this clause *could* fit on the report page being currently generated, the report group will be *forced* to appear on a new page.
3. The following points document the use for format 2 of the `LINE` clause:
   A. The column location of a screen section field is determined by the `COLUMN` (see [COLUMN], page 166) clause.

B. The value of *integer-1* must be 1 or greater.

C. If *identifier-1* is used to specify either an absolute or relative column position, *identifier-1* must be defined as a numeric item of any `USAGE` (see [USAGE], page 232) other than `COMPUTATIONAL-1` or `COMPUTATIONAL-2`, without editing symbols. The value of *identifier-1* at the time the screen data item is presented must be 1 or greater. Note that a `COMPUTATIONAL-1` or `COMPUTATIONAL-2` identifier will be accepted by the compiler, but will produce unpredictable results at run-time.

D. The screen line number upon which the data item containing this clause along with any subordinate data items will be displayed may be stated on an absolute basis (*i.e.* `LINE 5`) or on a relative basis based upon the previously-displayed line (*i.e.* `LINE PLUS 1`).

E. The symbol '`+`' may be used in lieu of the word `PLUS`, if desired; if '`+`' is used, however, there must be at least one space separating it from *integer-1*. Failure to include this space will cause the '`+`' to be simply treated as part of *integer-1* and will treat the `LINE` clause as an absolute line specification rather than a relative one.

F. If a screen data item's description includes the `FROM` (see [FROM], page 174), `TO` (see [TO], page 221), `USING` (see [USING], page 243) or `VALUE` (see [VALUE], page 244) clause but has no LINE clause, the "current screen line" will be assumed.

6.9.27B. LOWER

LOWER Attribute Syntax

```
LOWER
~~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause provides a means of explicitly stating that accepted data will be in lower case.

6.9.28. LOWLIGHT

LOWLIGHT Attribute Syntax

```
LOWLIGHT
~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

The `LOWLIGHT` clause controls the intensity of text (`FOREGROUND-COLOR`) by setting that intensity to its lowest of three possible settings.

1. This clause, along with `HIGHLIGHT` (see [HIGHLIGHT], page 178), are intended to provide a three-level intensity scheme (`LOWLIGHT` ... nothing (Normal) ... `HIGHLIGHT`). In environments such as a Windows console where only two levels of intensity are supported, `LOWLIGHT` is the same as leaving this clause off altogether.

See [Color Palette and Video Attributes], page 23, for more information on screen colors and video attributes.

6.9.29. NEXT GROUP

NEXT-GROUP Clause Syntax

```
NEXT GROUP IS { [ +|PLUS ] integer-2  }
~~~~ ~~~~~    {     ~~~~              }
              { NEXT|{NEXT PAGE}|PAGE }
                ~~~~  ~~~~ ~~~~  ~~~~
```

This syntax is valid in the following sections: `REPORT`

This clause defines any rules for where the next group to be presented on a report will begin, line-wise, with respect to the *last* line of the group in which this clause appears.

1. The reserved word `IS` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. The terms `NEXT`, `NEXT PAGE` and `PAGE` are interchangeable.
3. A report group must contain at least one `LINE NUMBER` clause in order to also contain a `NEXT GROUP` clause.
4. If the `RD` (see [REPORT SECTION], page 135) in which the report group containing a `NEXT GROUP` clause does not contain a `PAGE LIMITS` clause, only the `PLUS integer-1` option may be specified.
5. The `NEXT PAGE` option cannot be used in a `PAGE FOOTING`.
6. The `NEXT GROUP` option cannot be specified in either a `REPORT HEADING` or a `PAGE HEADING`.
7. The effects of `NEXT GROUP` will be in addition to any line spacing defined by the next-presented group's `LINE NUMBER` clause.

6.9.30. NO-ECHO

NO-ECHO Attribute Syntax

```
NO-ECHO
~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

The `NO-ECHO` clause will cause all data entered into the field to appear on the screen as asterisks.

1. The `NO-ECHO` and `SECURE` (see [SECURE], page 212) clauses are interchangeable, and may not be used together in the same data item description.
2. This clause may only be used on a field allowing data entry (a field containing either the `USING` (see [USING], page 243) or `TO` (see [TO], page 221) clause).

See [Color Palette and Video Attributes], page 23, for more information on screen colors and video attributes.

6.9.30B. NO UPDATE

NO-UPDATE Attribute Syntax

```
NO UPDATE
~~ ~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

The `NO UPDATE` clause forces the field to which this is applied will not be updated MORE??.

6.9.31. OCCURS

OCCURS Clause Syntax

```
GENERAL FORMAT.

 Format 1 (fixed-table):

 OCCURS integer-2 TIMES
 ~~~~~~

 Format 2 (occurs-depending-table):

 OCCURS [ integer-1 TO ] { integer-2 TIMES } [ DEPENDING ON identifier-1 ]
 ~~~~~~              ~~   { UNBOUNDED       }   ~~~~~~~~~

   [ ASCENDING|DESCENDING KEY IS identifier-5 ...  ]  ...
     ~~~~~~~~~ ~~~~~~~~~~
   [ INDEXED BY index-name-1 ...  ]
     ~~~~~~~

 REPORT SECTION.

 Format 3

 OCCURS [ integer-1 TO ] integer-2 TIMES  [ DEPENDING ON identifier-1 ]
 ~~~~~~              ~~                     ~~~~~~~~~

   [ STEP integer-3 ]
     ~~~~
   [ VARYING identifier-2 FROM { identifier-3 } BY { identifier-4 } ]
     ~~~~~~~              ~~~~ { integer-4    } ~~ { integer-5    }


 SCREEN SECTION.

 Format 4

 OCCURS integer-2 TIMES
 ~~~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`, `REPORT`, `SCREEN`

The `OCCURS` clause is used to create a data structure called a table, where entries in that structure repeat multiple times.

1. The reserved words `BY` (INDEXED), `IS`, `KEY`, `ON` and `TIMES` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The value of *integer-2* specifies how many entries will be allocated in the table.
3. The following is an example of how a table might be defined:

   ```
   05 QUARTERLY-REVENUE OCCURS 4 TIMES PIC 9(7)V99.
   ```

   This will allocate the following:

   ```
   QUARTERLY-REVENUE(1)
   QUARTERLY-REVENUE(2)
   QUARTERLY-REVENUE(3)
   QUARTERLY-REVENUE(4)
   ```

   Each occurrence is referenced using the subscript syntax (a numeric literal, arithmetic expression or numeric identifier enclosed within parenthesis) shown above.
4. The `OCCURS` clause may be used at the group level too, in which case the entire group structure repeats, as follows:

   ```
   05 GRP OCCURS 3 TIMES.
       10 A     PIC X(1).
       10 B     PIC X(1).
       10 C     PIC X(1).
   ```

   This would allow references to any of the following:

   `GRP(1)` includes A(1), B(1) and C(1)
   `GRP(2)` includes A(2), B(2) and C(2)
   `GRP(3)` includes A(3), B(3) and C(3)

   or each A,B,C item could be referenced as follows:

   `A(1)` character #1 of `GRP(1)`
   `B(1)` character #2 of `GRP(1)`
   `C(1)` character #3 of `GRP(1)`
   `A(2)` character #1 of `GRP(2)`
   `B(2)` character #2 of `GRP(2)`
   `C(2)` character #3 of `GRP(2)`
   `A(3)` character #1 of `GRP(3)`
   `B(3)` character #2 of `GRP(3)`
   `C(3)` character #3 of `GRP(3)`
5. The optional `DEPENDING ON` clause can be added to an `OCCURS` to create a variable-length table. In such cases, the value of *integer-1* specifies what the minimum number of entries in the table will be while *integer-2* specifies the maximum. Such tables will be allocated out to the maximum size specified as *integer-2*. At execution time the value of *identifier-1* will determine how many of the table elements are accessible.

6. See the documentation of the `SEARCH` (see [SEARCH], page 377), `SEARCH ALL` (see [SEARCH ALL], page 379) and `SORT` (see [SORT], page 391) statements for explanations of the `KEY` and `INDEXED BY` clauses.
7. The `OCCURS` clause cannot be specified in a data description entry that has a level number of 01, 66, 77, or 88, although it is valid in data items described *subordinate* to an 01-level data item.
8. The following points apply to an `OCCURS` used in the report section:
   A. The optional `STEP` clause defines an incrementation value that will be added to any absolute `LINE` (see [LINE], page 183) or `COLUMN` (see [COLUMN], page 166) number specifications that may be part of or subordinate to this data item's definition.
   B. The optional `VARYING` clause defines an identifier that may be used as a subscript for the multiple occurrences of this or any subordinate data item should the `SOURCE` (see [SOURCE], page 215) or `SUM` (see [SUM], page 520) clause(s) on this or subordinate data items reference entries within the table. The *identifier-2* data item is dynamically created as needed and cannot be referenced outside the scope of the report data item definition.
   C. The following two examples illustrate two different ways a report could include four quarters worth of sales figures in its detail lines — one doing things 'the hard way' and one using the advanced `OCCURS` capabilities of `STEP` and `VARYING`. Both assume the definition of the following table exists in working-storage:

```
05 SALES OCCURS 4 TIMES PIC 9(7)V99.
```

   First, the "Hard Way":

```
10 COL 7  PIC $(7)9.99 SOURCE SALES(1).
10 COL 17 PIC $(7)9.99 SOURCE SALES(2).
10 COL 27 PIC $(7)9.99 SOURCE SALES(3).
10 COL 37 PIC $(7)9.99 SOURCE SALES(4).
```

   And then using `STEP` and `VARYING`:

```
10 COL 7  OCCURS 4 TIMES STEP 10 VARYING QTR FROM 1 BY 1
          PIC $(7)9.99 SOURCE SALES(QTR).
```

6.9.32. OVERLINE

OVERLINE Attribute Syntax

```
OVERLINE
~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

The `OVERLINE` clause will introduce a horizontal line at the top edge of a screen field.

1. The `LEFTLINE` (see [LEFTLINE], page 181), `OVERLINE` and `UNDERLINE` (see [UNDERLINE], page 229) clauses may be used in any combination in a single field's description.
2. This clause is essentially non-functional when used within Windows command shell (cmd.exe) environments and running programs compiled using a GnuCOBOL implementation built using "PDCurses" (such as Windows/MinGW builds).
3. Whether or not this clause operates on Cygwin or UNIX/Linux/OSX systems will depend upon the video attribute capabilities of the terminal output drivers and "curses" software being used.

See [Color Palette and Video Attributes], page 23, for more information on screen colors and video attributes.

6.9.33. PICTURE

PICTURE Clause Syntax

```
PICTURE IS picture-string
~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`, `REPORT`, `SCREEN`

The picture clause defines the class (numeric, alphabetic or alphanumeric), size and format of the data that may be contained by the data item being defined. Sometimes this role is assisted by the `USAGE` (see [USAGE], page 232) clause, and in a few instances will be assumed entirely by that clause.

1. The reserved word `IS` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. The reserved word `PICTURE` may be abbreviated as `PIC`. Most programmers prefer to use the latter.
3. A picture clause may only be specified on an elementary item.
4. A *picture-string* is a sequence of the special symbols '`$`', '`*`', '`+`', '`,`', '`-`', '`.`', '`/`', '`0`' (zero), '`1`', '`9`', '`A`', '`B`', `CR`, `DB`, '`S`', '`V`', '`X`' and '`Z`'.
5. In general, each picture symbol represents either a single character in storage or a single decimal digit. There are a few exceptions, and they will be discussed as needed.
6. When a *picture-string* contains a repeated sequence of symbols — `PIC 9999/99/99` — for example, the repetition can be specified using a parenthetic repeat count, as in `PIC 9(4)/9(2)/9(2)`. Using repeat counts is optional and their use (or not) is entirely at the discretion of the programmer. Many programmers use repetition for small sequences (`PIC XXX`) and repeat counts for larger ones (`PIC 9(9)`.
7. This first set of picture symbols defines the basic data type of a data item. Each symbol represents a single character's worth of storage.

   '`A`' Defines storage reserved for a single alphabetic character ('`A`'-'`Z`', '`a`'-'`z`').

   '`N`' Defines storage reserved for a single character in the computer's *National Character set*. Support for national character sets in GnuCOBOL is currently only partially implemented, and the compile- and run-time effect of using the '`N`' picture symbol is the same as if `X(2)` had been coded, with the additional effect that such a field will qualify as a `NATIONAL` or `NATIONAL-EDITED` field on an `INITIALIZE` (see [INITIALIZE], page 339) statement.

   '`X`' Defines storage reserved for a single alphanumeric character (any character).

   '`9`' Defines storage reserved for a single numeric digit character ('`0`'-'`9`').

Typically, only one kind of each of those symbols is used in the same picture clause, but that isn't a requirement. Data items that, of the three symbols above, use nothing but '`A`' picture symbols are known as *Alphabetic Data Items* while those that use '`9`' picture symbols without any '`A`' or '`X`' symbols (or those that have a `USAGE` without a `PICTURE`) are known as *Numeric Data Items*. All other data items are referred to as *Alphanumeric Data Items*.

If you need to allocate space for a data item whose format is two letters followed by five digits followed by three letters, you could use the *picture-string* `AA99999AAA`, `A(2)9(5)A(3)` `XXXXXXXXXX` or `X(10)`. There is absolutely no functional difference whatsoever between the four — none of them provide any functionality the others do not. The first two probably make for better *documentation* of the expected field contents, but they don't provide any run-time enforcement capabilities.

As far as enforcement goes, however, both alphabetic and numeric picture strings do provide for both compile-time and run-time enforcement capabilities. In the case of compilation enforcement, the compiler can issue warning messages if you attempt to specify a non-numeric value for a numeric data item or if you attempt to `MOVE` (see [MOVE], page 352) a non-numeric data item to one that is numeric. Similar capabilities exist for alphabetic data items. At run-time, you may use a special class test (see [Class Conditions], page 46) to determine if the contents of a data item are entirely numeric or entirely alphabetic.

8. '`1`' Defines storage for a single bit representing a boolean condition with a states of zero or 1, condition of off or on. Warning the level of implementation of this feature is compiler version specific starting with v3.1-RC-1.
9. The following picture symbols may be used with numeric data items.

   '`P`' Defines an implied digit position that will be considered to be a zero when the data item is referenced at run-time. This symbol is used to allow data items that will contain very large values to be allocated using less storage by assuming a certain number of trailing zeros (one per '`P`') to exist at the end of values.

   The '`P`' symbol is not allowed in conjunction with '`N`'.

   The '`P`' symbol may only be used at the beginning or end of a picture clause.

   '`P`' is a repeatable symbol.

   All computations and `MOVE` (see [MOVE], page 352) operations involving such a data item will behave as if the zeros were actually there.

   For example, let's say you need to allocate a data item that contains however many millions of dollars of revenue your company has in gross revenues this year:

   ```
   01 Gross-Revenue PIC 9(9).
   ```

   In which case 9 characters of storage will be reserved. The values 000000000 through 999999999 will represent the gross-revenues. But, if only the millions are tracked (meaning the last six digits are always going to be 0), you could define the field as:

```
01 Gross-Revenue PIC 9(3)P(6).
```

Whenever Gross-Revenue is referenced in calculations, or whenever its value is moved to another data item, the value of Gross-Revenue will be treated as if it is *nnn*000000, where *nnn* is the actual value in storage.

If you wanted to store the value 128 million into that field, you would do so as if the 'P's were '9's:

```
MOVE 128000000 TO Gross-Revenue
```

A DISPLAY (see [DISPLAY], page 305) of a data item containing 'P' symbols is a little strange. The value displayed will be what is actually in storage, but the total size of the displayed value will be as if the 'P' symbols had been '9's. Thus, after the above statement established a value for Gross-Revenue, a DISPLAY Gross-Revenue would produce output of '000000128'.

'S' This symbol, if used, must be the very first symbol in the PICTURE value. A 'S' indicates that the data item is Signed, meaning that negative values are possible for this data item. Without an 'S', any negative values stored into this data item via a MOVE or arithmetic statement will have the negative sign stripped from it (in effect becoming the absolute value).

The 'S' symbol is not allowed in conjunction with 'N'.

The 'S' symbol may only occur once in a picture string. See [SIGN IS], page 213, for further discussion of how negative values may be stored in a numeric data item.

'V' This symbol is used to define where an implied decimal-point (if any) is located in a numeric item. Just as there may only be a single decimal point in a number so may there be no more than one 'V' in a PICTURE. Implied decimal points occupy no space in storage — they just specify how values are used. For example, if the value 1234 is in storage in a field defined as PIC 999V9, that value would be treated as 123.4 in any statements that referenced it.

The 'V' symbol is not allowed in conjunction with 'N'.

The 'V' symbol may only occur once in a picture string.

10. Any editing symbols introduced past this point will, if coded in the picture clause of an otherwise numeric data item, transform that data item from a numeric to a *Numeric Edited* data item. Numeric edited data items are treated as alphanumeric and may not serve either as table subscripts or as source arguments on an arithmetic statement.
11. The following are the fixed insertion editing symbols that may be specified in a picture string. Each of these editing symbols will insert a special character into the field value at the position it is specified in the picture string. These editing symbols will each introduce one extra character into the total field size for each occurrence of the symbol in the picture string.

'B' The 'B' editing symbol introduces a blank into the field value for each occurrence.

Multiple 'B' symbols may be coded.

The following example will format a ten digit number (presumably a telephone number) into a '### ### ####' layout:

```
...
    05 Phone-Number       PIC 9(3)B9(3)B9(4).
...
    MOVE 5185551212 TO Phone-Number
    DISPLAY Phone-Number
```

This code will display '518 555 1212'.

'0' The '0' (zero) editing symbol introduces one "0" character into the field value for each occurrence in the picture string.

Multiple '0' symbols may be coded.

Here's an example:

```
...
    05  Output-Item     PIC 909090909.
...
    MOVE 12345 TO Output-Item
    DISPLAY Output-Item
```

The above will display '102030405'.

'/' The '/' editing symbol inserts one "/" character into the field value for each occurrence in the picture string.

Multiple '/' symbols may be coded.

This editing symbol is most-frequently used to format dates, as follows:

```
...
    05  Year-Month-Day   PIC 9(4)/9(2)/9(2).
...
    MOVE 20140207 TO Year-Month-Day
    DISPLAY Year-Month-Day
```

This example displays '2014/02/07'.

12. The following are the numeric formatting symbols that may be specified in a picture string. Each of these editing symbols will insert special characters into the field value to present numbers in a "friendly" format. These editing symbols will each introduce one extra character into the total field size for each occurrence of the symbol in the picture string. Numeric fields whose picture clause contains these characters may neither be used as source fields in any calculation nor may they serve as source fields for the transfer of data values to any data item other than an alphanumeric field.

    '.' The '.' symbol inserts a decimal point into a numeric field value. When the contents of a numeric data item sending field are moved into a receiving data item whose picture clause contains the '.' editing symbol, implied ('V') or actual decimal point in the sending data item or literal, respectively, will be aligned with the '.' symbol in the receiving field. Digits are then transferred from the sending to the receiving field outward from the sending field's 'V' or '.', truncating sending digits if there aren't enough positions

in the receiving field. Any digit positions in the receiving field that don't receive digits from the sending field, if any, will be set to 0.

The '.' symbol is not allowed in conjunction with 'N'.

An example will probably help:

```
...
05  Source-Field   PIC 9(2)V9 VALUE 7.2.
05  Dest-Field     PIC 9(5).9(2).
...
MOVE 1234567.89 TO Dest-Field
DISPLAY Dest-Field
MOVE 19 TO Dest-Field
DISPLAY Dest-Field
MOVE Source-Field TO Dest-Field
DISPLAY Dest-Field
```

The example will display three results — '34567.89', '00019.00' and '00007.20'.

Both data item definitions *appear* to have *two* decimal points in their picture clauses. They actually don't, because the last character of every data item definition is always a period — the period that ends the definition.

',' The ',' symbol serves as a thousands separator. Many times, you'll see large numbers formatted with these symbols — for example, 123,456,789. This can be accomplished easily by adding thousands separator symbols to a picture string. Thousands separator symbols that aren't needed will behave as if they were '9's.

The ',' symbol is not allowed in conjunction with 'N'.

Here's an example:

```
...
05  My-Lottery-Winnings   PIC 9(3),9(3),9(3).
...
MOVE 12345 TO My-Lottery-Winnings
DISPLAY My-Lottery-Winnings
```

The value '0000012,345' (a very disappointing one for my retirement plans, but a good thousands separator demo) will be displayed. Notice how, since the first comma wasn't needed due to the meagre amount I won, it behaved like another '9'.

If desired, you may reverse the roles of the '.' and ',' editing symbols by specifying DECIMAL POINT IS COMMA in the SPECIAL-NAMES (see [SPECIAL-NAMES], page 92) paragraph.

13. The following are insertion symbols. They are used to insert an extra character (two in the case of CR and DB) to signify the sign (positive or negative) of the numeric value that is moved into the field whose picture string contains one of these symbols, or the fact that the data item represents a currency (money) amount. Only one of the '+', '-', CR or DB symbols may be used in a picture clause. In this context, when any of

these symbols are used in a *picture-string*, they must be at the end. The '+', '-' and/or currency symbols may also be used as floating editing symbols at the *beginning* of the *picture-string* — a subject that will be covered in the next numbered paragraph.

'+' If the value of the numeric value moved into the field is positive (0 or greater), a '+' character will be inserted. If the value is negative (less than 0), a '-' character is inserted.

The '+' symbol is not allowed in conjunction with 'N'.

'-' If the value of the numeric value moved into the field is positive (0 or greater), a space will be inserted. If the value is negative (less than 0), a '-' character is inserted.

The '-' symbol is not allowed in conjunction with 'N'.

CR This symbol is coded as the two characters 'C' and 'R'. If the value of the numeric value moved into the field is positive (0 or greater), two spaces will be inserted. If the value is negative (less than 0), the characters CR (credit) are inserted.

The CR symbol is not allowed in conjunction with 'N'.

DB This symbol is coded as the two characters 'D' and 'B'. If the value of the numeric value moved into the field is positive (0 or greater), two spaces will be inserted. If the value is negative (less than 0), the characters DB (debit) are inserted.

The DB symbol is not allowed in conjunction with 'N'.

'$' Regardless of the value moved into the field, this symbol will insert the currency symbol into the data item's value in the position where it occurs in the *picture-string* (see [SPECIAL-NAMES], page 92).

The '$' symbol is not allowed in conjunction with 'N'.

14. These editing symbols are known as floating replacement symbols. These symbols may occur in sequences *before* any '9' editing symbols in the *picture-string* of a numeric data item. Using these symbols transforms that numeric data item into a numerid *edited* data item, which can no longer be used in calculations or subscripts.

15. Each of the following symbols behave like a '9', until such point as all digits in the numeric value are exhausted and leading zeros are about to be inserted. In effect, these editing symbols define what should happen to those leading zero.

'$' Of those currency symbols that correspond to character positions in which leading zeros reside, the right-most will have its '0' value replaced by the currency symbol in-effect for the program (see [SPECIAL-NAMES], page 92). Any remaining leading zero values occupying positions described by this symbol will be replaced by spaces.

The '$' symbol is not allowed in conjunction with 'N'.

Any currency symbol coded to the right of a '.' will be treated exactly like a '9'.

'*' This symbol is referred to as a check protection symbol. All check-protection symbols that correspond to character positions in which leading zeros reside will have their '0' values replaced by '*'.

The '*' symbol is not allowed in conjunction with 'N'.

Any check-suppression symbol coded to the right of a '.' will be treated exactly like a '9'.

'+' Of those '+' symbols that correspond to character positions in which leading zeros reside, the right-most will have its '0' value replaced by a '+' if the value in the data item is zero or greater or a '-' otherwise. Any remaining leading zero values occupying positions described by this symbol will be replaced by spaces. You cannot use both '+' and '-' in the same *picture-string*.

The '+' symbol is not allowed in conjunction with 'N'.

Any '+' symbol coded to the right of a '.' will be treated exactly like a '9'.

'-' Of those '-' symbols that correspond to character positions in which leading zeros reside, the right-most will have its '0' value replaced by a space if the value in the data item is zero or greater or a '-' otherwise. Any remaining leading zero values occupying positions described by this symbol will be replaced by spaces. You cannot use both '+' and '-' in the same *picture-string*.

The '-' symbol is not allowed in conjunction with 'N'.

Any '-' symbol coded to the right of a '.' will be treated exactly like a '9'.

'Z' All 'Z' symbols that correspond to character positions in which leading zeros reside will have their '0' values replaced by spaces.

Any zero-suppression symbol coded to the right of a '.' will be treated exactly like a '9'.

'Z' and '*' should not be coded in the same *picture-string*

'+' and '-' should not be coded in the same *picture-string*

When multiple floating symbols are coded, even if there is only one of them used they will all be considered floating and will all be able to assume each other's properties. For example, if a data item has a `PIC +$ZZZZ9.99` *picture-string*, and a value of 1 is moved to that field at run-time, the resulting value will be (the *b* symbol represents a space) *bbbb*`+$1.00`. This is not consistent with many other COBOL implementations, where the result would have been `+$`*bbbb*`1.00`.

Most other COBOL implementations reject the use of multiple occurrences of multiple floating editing symbols. For example, they would reject *picture-string*s such as `+++$$$9.99`, `$$$ZZZ9.99` and so on. GnuCOBOL accepts these. Programmers creating GnuCOBOL programs should avoid such *picture-string*s if there is any likelihood that those programs may be used with other COBOL implementations.

6.9.34. PRESENT WHEN

PRESENT-WHEN Clause Syntax

```
PRESENT WHEN condition-name
~~~~~~~ ~~~~
```

This syntax is valid in the following sections: `REPORT`

This clause names an existing `Condition Name` (see [Condition Names], page 45) that will serve as a switch controlling the presentation or suppression of a report group.

1. If the specified condition-name has a value of FALSE when a `GENERATE` statement (see [GENERATE], page 332) causes a report group to be presented, the presentation of that group will be suppressed.
2. If the condition-name has a value of `TRUE`, the group will be presented.
3. See [Condition Names], page 45, for more information.

6.9.35. PROMPT

PROMPT Clause Syntax

```
PROMPT [ CHARACTER IS literal-1 | identifier-1 ]
~~~~~~   ~~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause defines the character that will be used as the fill-character for any input fields on the screen.

1. The reserved word `IS` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. The default prompt character, should no `CHARACTER` specification be coded, or should the `PROMPT` clause be absent altogether, is an underscore ('`_`').
3. Prompt characters will be automatically transformed into spaces upon input.

See [Color Palette and Video Attributes], page 23, for more information on screen colors and video attributes.

6.9.36. PROTECTED

PROTECTED Attribute Syntax

```
PROTECTED SIZE IS { identifier }
~~~~~~~~  ~~~~    { integer    }
```

This syntax is valid in the following sections: SCREEN

The PROTECTED extended clause will effect the specified field to be limited in size, regardless of the picture size.[1]

1. The SIZE phrase specifies the size (length) of the field. After the ACCEPT or DISPLAY is finished, the cursor is placed immediately after the field defined by this clause, unless this would place the cursor outside of the current terminal window. In this case, the cursor is wrapped around to the beginning of the next line (scrolling the window if necessary).
2. If the SIZE phrase is not used, then the field length defaults to the size of the item being accepted or displayed. If the CONVERT phrase is used, however, then the size of the field depends on the data type of the item and the verb being used.
   A. If the DISPLAY verb is executing, then the size is the same as if the CONVERT phrase were not specified except for numeric items. For numeric items, the size is the number of digits in the item, plus one if it is not an integer, plus one if it is signed. The remaining cases cover the size when an ACCEPT statement is used.
   B. If the item is numeric or numeric edited, then the size is the number of digits in the item, plus one if it is not an integer, plus one if it is signed.
   C. If the item is alphanumeric edited, then the size is set to the number of 'A' or 'X' positions specified in its PICTURE clause.
   D. For all other data types, the field size is set to the size of the item (same as if CONVERT were not specified).
3. Note that the OUTPUT phrase changes the way in which the default field size is computed. See that heading above for details. Also note that the OUTPUT phrase affects only the way items are displayed on the screen; the internal format of accepted data is not affected.
4. Note that you cannot supply the CONVERT phrase in the Screen Section. Thus the size of a Screen Section field is always the size of its screen entry unless the SIZE phrase is specified.

[1] OR DOES IT? *author uncertain*

6.9.37. REDEFINES

REDEFINES Clause Syntax

```
REDEFINES identifier-1
~~~~~~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`

The `REDEFINES` clause causes the data item in who's definition the `REDEFINES` clause is specified (hereafter referred to as the redefines object) to occupy the same physical storage space as *identifier-1* (hereafter referred to as the redefines subject).

1. The following rules must all be followed in order to use `REDEFINES`:
   A. The level number of both the subject and object data items must be the same.
   B. The level numbers of both the subject and object data items cannot be 66, 78 or 88.
   C. If *N* represents the level number of the object, then no other data items with level number *N* may be defined between the subject and object data items unless they too are `REDEFINES` of the subject.
   D. If *N* represents the level number of the object, then no other data items with a level number numerically less than *N* may be defined between the subject and object data items.
   E. The total allocated size of the subject data item must be the same as the total allocated size of the object data item.
   F. No `OCCURS` (see [OCCURS], page 190) clause may be part of the definition of either the subject or object data items. Either or both, however, may be group items that *contain* data items with `OCCURS` clauses.
   G. No `VALUE` (see [VALUE], page 244) clause may be defined on the object data item, and no data items subordinate to the object data item may have `VALUE` clauses, with the exception of level-88 condition names.

6.9.38. RENAMES

RENAMES Clause Syntax

```
RENAMES identifier-1 [ THRU|THROUGH identifier-2
~~~~~~~                ~~~~ ~~~~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`

The `RENAMES` clause regroups previously defined items by specifying alternative, possibly overlapping, groupings of elementary data items.

1. The reserved words `THRU` and `THROUGH` are interchangeable.
2. You must use the level number 66 for data description entries that contain the `RENAMES` clause.
3. The *identifier-1* and *identifier-2* data items, along with all data items defined between those two data items in the program source, must all be contained within the same 01-level record description.
4. See [66-Level Data Items], page 150, for additional information on the `RENAMES` clause.

6.9.39. REQUIRED

REQUIRED Attribute Syntax

```
REQUIRED
~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause forces the user to enter data into the field it is specified on (or into all subordinate input-capable fields if `REQUIRED` is specified on a group item).

1. The `EMPTY-CHECK` (see [EMPTY-CHECK], page 169) and `REQUIRED` clauses are interchangeable, and may not be used together in the same data item description.
2. In order to take effect, the user must first move the cursor into the field having this clause in its definition.
3. The `ACCEPT data-item` statement (see [ACCEPT data-item], page 272) will ignore the Enter key and any other cursor-moving keystrokes that would cause the cursor to move to another screen item *unless* data has been entered into the field. Function keys will still be allowed to terminate the `ACCEPT`.
4. In order to be functional, this attribute must be supported by the underlying "curses" package your GnuCOBOL implementation was built with. As of this time, the "PDCurses" package (used for native Windows or MinGW builds) does not support `REQUIRED`.

6.9.40. REVERSE-VIDEO

REVERSE-VIDEO Attribute Syntax

```
REVERSE-VIDEO
~~~~~~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

The `REVERSE-VIDEO` attribute swaps the specified or implied `FOREGROUND-COLOR` (see [FOREGROUND-COLOR], page 173) and `BACKGROUND-COLOR` (see [BACKGROUND-COLOR], page 158) attributes for the field whose definition contains this clause (or all subordinate fields if used on a group item).

See [Color Palette and Video Attributes], page 23, for more information on screen colors and video attributes.

6.9.41. SAME AS

SAME-AS Clause Syntax

```
SAME AS data-name-1
~~~~ ~~
```

The syntax is valid in the following sections: `WORKING-STORAGE`

The `SAME AS` causes a data item to inherit the same definition of another data item.

1. Data-name-1 is a data Item defined elsewhere in the same program.
2. Data-name-1 shall not be subscripted.
3. A data description entry that specifies the SAME AS clause shall not be immediately followed by a subordinate data description entry or level 88 entry.
4. Neither the description of data-name-1 nor the description of any data items subordinate to the subject of the entry shall directly or indirectly contain a SAME AS clause that references the subject of the entry or any group item to which this entry is subordinate.
5. The description of data-name-1, including its subordinate data items, shall not contain a TYPE clause that references the record to which this entry is subordinate.
6. The description of data-name-1 shall not contain an OCCURS clause. However, items subordinate to data-name-1 may contain OCCURS clauses.
7. Data-name-1 shall reference an elementary item or a level 1 group item described in the file, working-storage, local-storage, or linkage section.
8. If the subject of the entry is a level 77 item, data-name-1 shall reference an elementary item.
9. A group item to which the subject of the entry is subordinate shall not contain a GROUP-USAGE, SIGN, or USAGE clause.
10. The effect of the SAME AS clause is as though the data description identified by data-name-1 had been coded in place of the SAME AS clause, excluding the level number, name, and the EXTERNAL, GLOBAL, REDEFINES clauses specified for data-name-1; level numbers of subordinate items may be adjusted as described in general rule 2.
11. If data-name-1 describes a group item:
12. the subject of the entry is a group whose subordinate elements have the same names, descriptions, and hierarchy as the subordinate elements of data-name-1, the level numbers of items subordinate to that group are adjusted, if necessary, to preserve the hierarchy of data-name-1, level numbers in the resulting hierarchy may exceed 49.

```
IDENTIFICATION   DIVISION.
PROGRAM-ID.      prog.
DATA             DIVISION.
```

```
WORKING-STORAGE  SECTION.
01 MESSAGE-TEXT-2 EXTERNAL.
     02 MAIN-FILE-NAME     PIC X(50).
     02 FILLER REDEFINES MAIN-FILE-NAME.
         05 FILLER PIC 9999.
     02 MAIN-FILE-NAME-2.
         05 FILLER PIC 9999.
         05 DETAIL-NO PIC 9999.
    02 FILLER         SAME AS MAIN-FILE-NAME.
77 OUTPUT-NAME     SAME AS DETAIL-NO GLOBAL.
01 Z-MESSAGE-T2    SAME AS MAIN-FILE-NAME-2.
01 Z-MESSAGE-T3.
      49 MT3                    SAME AS MESSAGE-TEXT-2.
      49 MT3-REN  REDEFINES MT3 SAME AS MESSAGE-TEXT-2.

PROCEDURE DIVISION.
    DISPLAY MAIN-FILE-NAME OF MESSAGE-TEXT-2
    DISPLAY DETAIL-NO      OF Z-MESSAGE-T2
    DISPLAY MAIN-FILE-NAME OF MT3
    DISPLAY OUTPUT-NAME
    GOBACK.
```

6.9.41B. SCROLL DOWN

SCROLL-DOWN Attribute Syntax

```
SCROLL DOWN
~~~~~~ ~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause will cause downward scrolling.

6.9.41C. SCROLL UP

SCROLL-UP Attribute Syntax

```
SCROLL UP
~~~~~~ ~~
```

This syntax is valid in the following sections: `SCREEN`

This clause will cause upward scrolling. MORE HERE ??

6.9.42. SECURE

SECURE Attribute Syntax

```
SECURE
~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause will cause all data entered into the field to appear on the screen as asterisks.

1. The `NO-ECHO` (see [NO-ECHO], page 188) and `SECURE` clauses are interchangeable, and may not be used together in the same data item description.
2. This clause may only be used on a field allowing data entry (a field containing either the `USING` (see [USING], page 243) or `TO` (see [TO], page 221) clause).

See [Color Palette and Video Attributes], page 23, for more information on screen colors and video attributes.

6.9.43. SIGN IS

SIGN-IS Clause Syntax

```
SIGN IS LEADING|TRAILING [ SEPARATE CHARACTER ]
~~~~    ~~~~~~~ ~~~~~~~~   ~~~~~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`, `REPORT`, `SCREEN`

This clause, allowable only for `USAGE DISPLAY` numeric data items, specifies how an '`S`' symbol will be interpreted in a data item's picture clause.

1. The reserved words `CHARACTER` and `IS` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. *Without* the `SEPARATE CHARACTER` option, the sign of the data item's value will be encoded by transforming the last (see `TRAILING`) or first (see `LEADING`) digit as follows:

First/Last Digit	Value For Positive	Value for Negative
0	0	p
1	1	q
2	2	r
3	3	s
4	4	t
5	5	u
6	6	v
7	7	w
8	8	x
9	9	y

3. If the `SEPARATE CHARACTER` clause *is* used, then an actual '`+`' or '`-`' character will be inserted into the field's value as the first (`LEADING`) or last (`TRAILING`) character. Note that having this character embedded within the data item's storage does not prevent the data item from being used as a source field in arithmetic operations.
4. When `SEPARATE CHARACTER` is specified, the '`S`' symbol in the data item's `PICTURE` must be counted when determining the data item's size.
5. Neither the presence of an encoded digit (see above) nor an actual '`+`' or '`-`' character embedded within the data item's storage prevents the data item from being used as a source field in arithmetic operations.

6.9.43B. SIZE

SIZE Clause Syntax

```
SIZE
~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause does WHAT EXACTLY ??????

6.9.44. SOURCE

SOURCE Clause Syntax

```
SOURCE IS literal-1 | identifier-1 [ ROUNDED ]
~~~~~~                              ~~~~~~~
```

This syntax is valid in the following sections: `REPORT`

This clause logically attaches a report section data item to another data item defined elsewhere in the data division.

1. The reserved word `IS` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. When the report group containing this clause is presented, the value of the specified numeric literal or identifier will be automatically moved to the report data item prior to presentation.
3. The specified identifier may be defined anywhere in the data division, but if it is defined in the report section it may only be `PAGE-COUNTER`, `LINE-COUNTER` or a `SUM` (see [SUM], page 520) counter.
4. The `PICTURE` (see [PICTURE], page 194) of the report data item must be such that it would be legal to `MOVE` (see [MOVE], page 352) the specified literal or identifier to a data item with that `PICTURE`.
5. The `ROUNDED` option comes into play should the number of digits to the right of an actual or assumed decimal point be different between the specified literal or identifier value (the "source value") and the `PICTURE` specified for the field in whose definition the `SOURCE` clause appears (the "target field"). *Without* `ROUNDED`, excess digits in the source value will simply be truncated to fit the target field. *With* `ROUNDED`, the source value will be arithmetically rounded to fit the target field. See [ROUNDED], page 261, for information on the `NEAREST-AWAY-FROM-ZERO` rounding rule, which is the one that will apply.

6.9.45. SUM OF

SUM-OF Clause Syntax

```
SUM OF { identifier-7 }...  [ { RESET ON FINAL|identifier-8 } ]
~~~    { literal-2    }      { ~~~~~    ~~~~~              }
                             { UPON identifier-9           }
                               ~~~~
```

This syntax is valid in the following sections: `REPORT`

The `SUM` clause establishes a summation counter whose value will be arithmetically calculated whenever the field is presented.

1. The reserved words `OF` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The `SUM` clause may only appear in a `CONTROL FOOTING` report group.
3. If the data item in which the `SUM` clause appears has been assigned its own identifier name, and that name is not `FILLER`, then that data item is referred to as a sum counter.
4. All *identifier-7* data items must be non-edited numeric in nature.
5. If any *identifier-7* data item is defined in the report section, it must be a sum counter.
6. Any *identifier-7* data items that are sum counters must either be defined in the same report group as the data item in which this `SUM` clause appears or they must be defined in a report data item that exists at a lower level in this report's control hierarchy. See [Control Hierarchy], page 609, for additional information.
7. The `PICTURE` of the report data item in who's description this `SUM` clause appears in must be such that it would be legal to `MOVE` (see [MOVE], page 352) the specified *identifier-7* or *literal-2* value to a data item with that `PICTURE`.
8. The following points apply to the `UPON` option:
   A. The data item *identifier-9* must be the name of a detail group specified in the same report as the control footing group in which this `SUM` clause appears.
   B. The presence of an `UPON` clause limits the `SUM` clause to adding the specified numeric literal or identifier value into the sum counter only when a `GENERATE` *identifier-9* statement is executed.
   C. If there is no `UPON` clause specified, the value of *identifier-7* or *literal-2* will be added into the sum counter whenever a `GENERATE` (see [GENERATE], page 332) of any detail report group in the report is executed.
   D. If there is only a single detail group in the report's definition, the `UPON` clause is meaningless.
9. The following points apply to the `RESET` option:
   A. If the `RESET` option is coded, `FINAL` or *identifier-8* (whichever is coded on the `RESET`) must be one of the report's control breaks specified on the `CONTROLS` clause.

B. If there is no `RESET` option coded, the sum counter will be reset back to zero after each time the control footing containing the `SUM` clause is presented. This is the typical behaviour that would be expected.

C. If, however, you want to reset the `SUM` counter only when the control footing for a control break higher in the control hierarchy is presented, specify that higher control break on the `RESET` option.

6.9.46. SYNCRONIZED

SYNCRONIZED Syntax

```
SYNCRONIZED|SYNCHRONISED [ LEFT|RIGHT ]
~~~~         ~~~~            ~~~~ ~~~~~
```

The `LEFT` and `RIGHT` (SYNCRONIZED) clauses are syntactically recognized but are otherwise non-functional.

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`

This optional clause optimizes the storage of binary numeric items to store them in such a manner as to make it as fast as possible for the CPU to fetch them.

1. The reserved words `SYNCRONIZED` and `SYNCHRONISED` are interchangeable, and may be abbreviated as `SYNC`.

2. If the `SYNCRONIZED` clause is coded on anything but a numeric data item with a `USAGE` (see [USAGE], page 232) that specifies storage of data in a binary form, the `SYNCRONIZED` clause will be ignored.

3. Synchronization is performed (by the compiler) as follows:

   A. If the binary item occupies one byte of storage, no synchronization is performed.

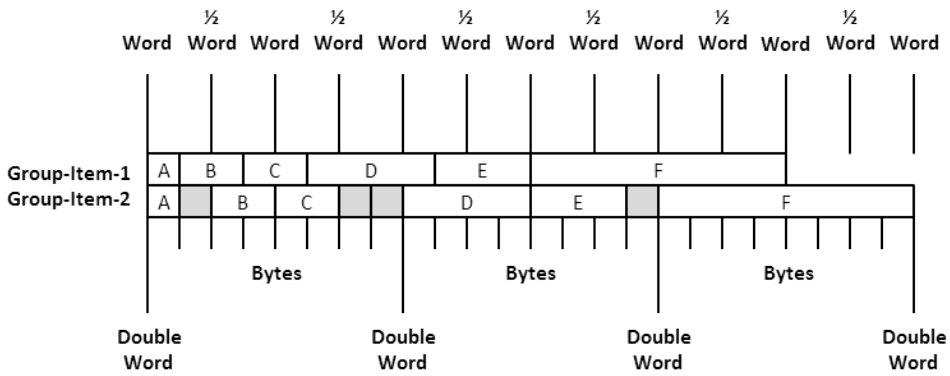
   B. If the binary item occupies two bytes of storage, the binary item is allocated at the next half-word boundary.

   C. If the binary item occupies four bytes of storage, the binary item is allocated at the next word boundary.

   D. If the binary item occupies eight bytes of storage, the binary item is allocated at the next word boundary.

The following illustrates the allocation of a group of data items both without and with the `SYNCRONIZED` option. The grey blocks represent the unused bytes that are allocated in the Group-Item-2 structure because of the `SYNC` clauses.

```
01  Group-Item-1.                          01  Group-Item-2.
    05 A            PIC X(1).                  05 A            PIC X(1).
    05 B            USAGE BINARY-SHORT.        05 B    SYNC    USAGE BINARY-SHORT.
    05 C            PIC X(2).                  05 C            PIC X(2).
    05 D            USAGE BINARY-LONG.         05 D    SYNC    USAGE BINARY-LONG.
    05 E            PIC X(3).                  05 E            PIC X(3).
    05 F            USAGE BINARY-DOUBLE.       05 F    SYNC    USAGE BINARY-DOUBLE.
```



6.9.46B. TIME OUT

TIME-OUT Clause Syntax

```
TIME OUT
~~~~ ~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause will force a specific time for the data tp be entered.

6.9.47. TO

TO Clause Syntax

```
TO identifier-5
~~
```

This syntax is valid in the following sections: `SCREEN`

This clause logically attaches a screen section data item to another data item defined elsewhere in the data division.

1. The `TO` clause is used to define a data-entry field with no initial value; when a value is entered, it will be saved to the specified identifier.
2. The `FROM` (see [FROM], page 174), `TO`, `USING` (see [USING], page 243) and `VALUE` (see [VALUE], page 244) clauses are mutually-exclusive in any screen section data item's definition.

6.9.48. TYPE

TYPE Clause Syntax

```
Format 1:  All Sections other than REPORT

[ TYPE TO type-name-1 ]
  ~~~~


Format 2:    REPORT SECTION Only

[ TYPE IS { RH|{REPORT HEADING}                       } ]
  ~~~~    { ~~  ~~~~~~ ~~~~~~~                       }
          { PH|{PAGE HEADING}                         }
          { ~~  ~~~~ ~~~~~~~                         }
          { CH|{CONTROL HEADING} FINAL|identifier-2    }
          { ~~  ~~~~~~~ ~~~~~~~  ~~~~~                }
          { DE|DETAIL                                 }
          { ~~ ~~~~~~                                 }
          { CF|{CONTROL FOOTING} FINAL|identifier-2    }
          { ~~  ~~~~~~~ ~~~~~~~  ~~~~~                }
          { PF|{PAGE FOOTING}                         }
          {  ~~ ~~~~ ~~~~~~~                          }
          { RF|{REPORT FOOTING}                       }
            ~~  ~~~~~~ ~~~~~~~
```

The syntax is valid in all other sections than: `REPORT`

As format 1

1. The TYPE clause indicates that the data description of the subject of the entry is specified by a user-defined data type.
2. The user defined data type is defined using the TYPEDEF clause, which is described in TYPEDEF clause `TYPEDEF` (see [TYPEDEF], page 227).
3. The following general rules apply: If type-name-1 (defined using the TYPEDEF clause) describes a group item, then the subject of the TYPE clause is a group item whose subordinate elements have the same names, descriptions, and hierarchies as the subordinate elements of type-name-1.
4. Since the subject of the TYPE clause may have a level number as high as 49 and type-name-1 may be a group item with 49 levels, the number of levels of this hierarchy may exceed 49. In fact, since descriptions of type names may reference other type names, there is no limit to the number of levels in this hierarchy.
5. If a VALUE clause is specified in the data description of the subject of the TYPE

clause, any VALUE clause specified in the description of type-name-1 is ignored for this entry.

6. The scoping rules for type names are similar to the scoping rules for data names.
7. Reference modification is not allowed for an elementary item that is the subject of a TYPE clause.
8. The description of type-name-1, including its subordinate data items, cannot contain a TYPE clause that references the record to which the subject of the TYPE clause (that references type-name-1), is subordinate.
9. For example, A is a group item defined using the TYPEDEF clause. B is also a group item defined using the TYPEDEF clause, but which also includes a subordinate item of TYPE A. This being the case, the type definition for A cannot include items of TYPE B.
10. The subject of a TYPE clause cannot be renamed in whole, or in part and cannot be redefined explicitly or implicitly.
11. If the subject of a TYPE clause is subordinate to a group item, the data description of the group item cannot contain the USAGE clause.
12. The TYPE clause cannot occur in a data description entry with the BLANK WHEN ZERO, FORMAT, JUSTIFIED, PICTURE, REDEFINES, RENAMES, SIGN, SYNCHRONIZED, or USAGE clause.
13. The TYPE clause can be specified in a data description entry with the EXTERNAL, GLOBAL, OCCURS, TYPEDEF, and VALUE clauses.
14. The essential characteristics of a type, which is identified by its type-name, are the relative positions and lengths of the elementary items defined in the type declaration, and the BLANK WHEN ZERO, JUSTIFIED, PICTURE, SIGN, SYNCHRONIZED, and USAGE clauses specified for each of these elementary items.
15. The TYPE clause shall not be specified in the same data description entry with any clauses except BASED, CLASS, CONSTANT RECORD, DEFAULT, DESTINATION, entry-name, EXTERNAL, GLOBAL, INVALID, level-number, OCCURS, PRESENT WHEN, PROPERTY, TYPEDEF, VALIDATE-STATUS, VALUE, and VARYING.
16. Figure 1. Example Showing How TYPEDEF and TYPE Clauses Can Be Used

```
IDENTIFICATION   DIVISION.
PROGRAM-ID.      prog.
DATA             DIVISION.
WORKING-STORAGE  SECTION.
01 MAIN-FILE-NAME-T     PIC X(50) IS TYPEDEF.
01 DETAIL-NO-T             PIC 9999  IS TYPEDEF.
 01 MAIN-FILE-NAME-2T              IS TYPEDEF.
   05 FILLER    PIC 9999.
   05 DETAIL-NO TYPE TO DETAIL-NO-T.

01  MESSAGE-TEXT-2T IS TYPEDEF.
  02 MAIN-FILE-NAME     TYPE MAIN-FILE-NAME-T.
```

```
  02 FILLER REDEFINES MAIN-FILE-NAME.
      05 FILLER PIC 9999.
  02 MAIN-FILE-NAME-2   TYPE MAIN-FILE-NAME-2T.
  02 FILLER               TYPE MAIN-FILE-NAME-T.

01  MESSAGE-TEXT-2 EXTERNAL TYPE MESSAGE-TEXT-2T.
77 OUTPUT-NAME TYPE TO DETAIL-NO-T GLOBAL.
01 Z-MESSAGE-T2 TYPE MAIN-FILE-NAME-2T.
01 Z-MESSAGE-T3.
   49 MT3                    TYPE MESSAGE-TEXT-2T.
   49 MT3-REN  REDEFINES MT3 TYPE MESSAGE-TEXT-2T.

PROCEDURE DIVISION.
    DISPLAY MAIN-FILE-NAME OF MESSAGE-TEXT-2
    DISPLAY DETAIL-NO         OF Z-MESSAGE-T2
    DISPLAY MAIN-FILE-NAME OF MT3
    DISPLAY OUTPUT-NAME
    GOBACK.
```

17. Figure 2. Example Showing How TYPEDEF and TYPE Clauses Can Be Used

```
 DATA DIVISION.
 FILE SECTION.
 FD PRT-FILE.
 01 PRT-REC.
    05 PRT-RECORD                PIC X(132).
 01 INVE-TYP-T IS TYPEDEF        PIC S9(3) VALUE 0.
         88 INVE-TYP-BOOK             VALUE 4, 5.
    88 INVE-TYP-BOOK-001         VALUE 4.
    88 INVE-TYP-BOOK-002         VALUE 5.
    88 INVE-TYP-CLOTHES          VALUE 1, 2, 3.
    88 INVE-TYP-CLOTHES-SWEATERS VALUE 1.
    88 INVE-TYP-CLOTHES-SOCKS    VALUE 2.
    88 INVE-TYP-CLOTHES-PANTS    VALUE 3.

FD DATA-IN.
01 DATA-IN-REC.
   05 INVE-TYP          TYPE INVE-TYP-T.
   05 FILLER            PIC X(80).

WORKING-STORAGE SECTION.
01 ARTI-PRICE-T TYPEDEF PIC S9(4)V9(2) value 0.
01 ARTI-COLOR-T TYPEDEF PIC S9(2) VALUE 1.
   88 ARTI-COLOR-BLUE   VALUE 1.
   88 ARTI-COLOR-RED    VALUE 2.
```

```
   88 ARTI-COLOR-GREEN  VALUE 3.

01 ARTI-SIZE-T    TYPEDEF PIC S9(2) VALUE 10.
01 ARTI-COUNTER-T TYPEDEF PIC S9(6) VALUE 0.

01 ARTI-B-T TYPEDEF.
   05 ARTI-B-VALUE PIC s9(2).
      88 ARTI-B-BLUE  VALUE 1.
      88 ARTI-B-RED   VALUE 2.
      88 ARTI-B-GREEN VALUE 3.

01 TEST-ARTI       TYPE ARTI-B-T.

01 WORK-INVE-TYP  TYPE INVE-TYP-T.

01 CLOTHING-ARTI IS TYPEDEF.
   05 CLOTHING-TYP    TYPE INVE-TYP-T.
   05 PRICE           TYPE ARTI-PRICE-T.
   05 COLOR           TYPE ARTI-COLOR-T.
   05 CLOTHING-SIZE   TYPE ARTI-SIZE-T.
   05 FILLER PIC X(70).

01 SWEATERS TYPE CLOTHING-ARTI.
01 SOCKS    TYPE CLOTHING-ARTI.
01 PANTS    TYPE CLOTHING-ARTI.

01 BOOK-ARTI IS TYPEDEF.
   05 BOOK-TYP     TYPE INVE-TYP-T.
   05 PRICE        TYPE ARTI-PRICE-T.
   05 FILLER       PIC X(20).
   05 BOOK-TITLE   PIC X(40).
   05 FILLER       PIC X(14).

01 BOOK-001       TYPE BOOK-ARTI.
01 BOOK-002       TYPE BOOK-ARTI.

01 SWEATERS-COUNT TYPE ARTI-COUNTER-T.
01 SOCKS-COUNT    TYPE ARTI-COUNTER-T.
01 PANTS-COUNT    TYPE ARTI-COUNTER-T.

01 BOOK-001-COUNT TYPE ARTI-COUNTER-T.
01 BOOK-002-COUNT TYPE ARTI-COUNTER-T.

01 CLOTHES-COUNT  TYPE ARTI-COUNTER-T.
01 BOOK-COUNT     TYPE ARTI-COUNTER-T.
```

This syntax is valid in the following sections: `REPORT`

As Format 2

1. This clause defines the type of report group that is being defined for a report.
2. This clause is required on any 01-level data item definitions (other than 01-level constants) in the report section. This clause is invalid on any other report section data item definitions.
3. There may be a maximum of one (1) report group per `RD` defined with a `TYPE` of `REPORT HEADING`, `PAGE HEADING`, `PAGE FOOTING` and `REPORT FOOTING`.
4. There must be either a `CONTROL HEADING` or a `CONTROL FOOTING` or both specified for each entry specified on the `CONTROLS ARE` clause of the `RD`.
5. The various report groups that constitute a report may be defined in any order.
6. See [RWCS Lexicon], page 605, for a description of the seven different types of report groups.

6.9.49. TYPEDEF

TYPEDEF Clause Syntax

```
01  data-name-1 IS TYPEDEF
                 ~~ ~~~~~~
```

This syntax is valid in the following sections FILE, WORKING-STORAGE, LOCAL-STORAGE and LINKAGE

1. The TYPEDEF clause identifies a type declaration which creates a user defined data type and is used to apply this user defined data type to the description of a data item.
2. The TYPEDEF clause specifies that the data description entry is a type declaration.
3. The TYPEDEF clause can only be specified for level 01 entries, which can also be group items, and for which the data-name format of the entry name clause is specified. If the TYPEDEF clause is specified for a data description, then that same data description must include a data-name, that is, it must not be specified with either an implicit or explicit FILLER clause.
4. If a group item is specified, all subordinate items of the group become part of the type declaration.
5. No storage is allocated for a type declaration.
6. These type definitions act like templates that can then be used to define new data items using the TYPE clause or that can subsequently be referenced in a USAGE clause.
7. The new data item acquires all the characteristics of the user-defined data type. If the user defined data type is a group item, then the new data item has subordinate elements of the same name, description, and hierarchy as those belonging to the user defined data type.
8. All of the other data description clauses, if they are specified, are assumed by any data item that is defined using the user defined data type (within the TYPE or USAGE clause).
9. The following clauses cannot be specified along with TYPEDEF: EXTERNAL, GLOBAL, LIKE, OCCURS, REDEFINES.
10. If the TYPEDEF clause is specified for a group item, then subordinate items can be specified with OCCURS or REDEFINES clauses. TYPEDEF cannot be used with complex OCCURS DEPENDING ON. This means that you cannot specify an OCCURS DEPENDING ON clause within a table that is part of a TYPEDEF.
11. The VALUE clause cannot be specified either in the data descriptions specifying the TYPEDEF clause or in any subordinate item except for condition-names (88 level entries) within the TYPEDEF structure.
12. The TYPE clause can be specified in the same data description entry as the TYPEDEF clause but the description of the subject of the entry, including its subordinate items, shall not contain a TYPE clause that directly or indirectly references this type definition.

13. In FILE SECTION, a data description entry declared at level number 1 that includes a TYPEDEF clause is not a record description entry.

6.9.50. UNDERLINE

UNDERLINE Attribute Syntax

```
UNDERLINE
~~~~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

The `UNDERLINE` clause will introduce a horizontal line at the bottom edge of a screen field.

1. The `LEFTLINE` (see [LEFTLINE], page 181), `OVERLINE` (see [OVERLINE], page 193) and `UNDERLINE` clauses may be used in any combination in a single field's description.
2. This clause is essentially non-functional when used within Windows command shell (cmd.exe) environments and running programs compiled using a GnuCOBOL implementation built using "PDCurses" (such as Windows/MinGW builds).
3. Whether or not this clause operates on Cygwin or UNIX/Linux/OSX systems will depend upon the video attribute capabilities of the terminal output drivers and "curses" software being used.

See [Color Palette and Video Attributes], page 23, for more information on screen colors and video attributes.

6.9.50B. UPDATE

UPDATE Clause Syntax

```
UPDATE
~~~~~~
```

This syntax is valid in the following sections: `SCREEN`

The `UPDATE` clause will display the existing data before allowing user to update it.

6.9.50C. UPPER

UPPER Clause Syntax

```
UPPER
~~~~~
```

This syntax is valid in the following sections: `SCREEN`

The `UPPER` clause force the accepted data to be in Upper Case.

6.9.51. USAGE

USAGE Clause Syntax

```
USAGE IS data-item-usage
~~~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`, `REPORT`

The `USAGE` clause defines the format that will be used to store the value of a data item.

1. The reserved word `IS` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. The following table summarizes the various USAGE specifications available in GnuCOBOL.

   `BINARY`
   `~~~~~~`

Range of Values:	Defined by the quantity of '`9`'s and the presence or absence of an '`S`' in the `PICTURE`
Storage Format:	Compatible Binary Integer
Negative Values Allowed?:	If `PICTURE` contains '`S`'
`PICTURE` Used?:	Yes

   `BINARY-C-LONG [ SIGNED ]`
   `~~~~~~~~~~~~~`

3. Depending on the system hardware, the range is either 0 to 2^31 or 0 to 2^63.

   `BINARY-C-LONG UNSIGNED`
   `~~~~~~~~~~~~~ ~~~~~~~~`

4. Depending on the system hardware, the range is either 0 to 2^32 or 0 to 2^64. This USAGE should only be used for direct CALLs into C and otherwise should not be used (and it won't compile with any strict -std as it is a GnuCOBOL-only extension).

   `BINARY-CHAR [ SIGNED ]`
   `~~~~~~~~~~~`

Range of Values:	-128 – 127
Storage Format:	Native Binary Integer
Negative Values Allowed?:	Yes

`PICTURE` Used?: No

`BINARY-CHAR UNSIGNED`

Range of Values: 0 – 255

Storage Format: Native Binary Integer

Negative Values Allowed?: No

`PICTURE` Used?: No

`BINARY-DOUBLE [ SIGNED ]`

Range of Values: -9,223,372,036,854,775,808 – 9,223,372,036,854,775,807

Storage Format: Native Binary Integer

Negative Values Allowed?: Yes

`PICTURE` Used?: No

`BINARY-DOUBLE UNSIGNED`

Range of Values: 0 – 18,446,744,073,709,551,615

Storage Format: Native Binary Integer

Negative Values Allowed?: No

`PICTURE` Used?: No

`BINARY-INT`

Same as `BINARY-LONG SIGNED`

`BINARY-LONG [ SIGNED ]`

Range of Values: -2,147,483,648 – 2,147,483,647

Storage Format: Native Binary Integer

Negative Values Allowed?: Yes

`PICTURE` Used?: No

```
BINARY-LONG UNSIGNED
~~~~~~~~~~~ ~~~~~~~~
```

Range of Values: 0 – 4,294,967,295

Storage Format: Native Binary Integer

Negative Values Allowed?: No

`PICTURE` Used?: No

```
BINARY-LONG-LONG
~~~~~~~~~~~~~~~~
```

Same as `BINARY-DOUBLE SIGNED`

```
BINARY-SHORT [ SIGNED ]
~~~~~~~~~~~~
```

Range of Values: -32,768 – 32,767

Storage Format: Native Binary Integer

Negative Values Allowed?: Yes

`PICTURE` Used?: No

```
BINARY-SHORT UNSIGNED
~~~~~~~~~~~~ ~~~~~~~~
```

Range of Values: 0 – 65,535

Storage Format: Native Binary Integer

Negative Values Allowed?: No

`PICTURE` Used?: No

```
COMPUTATIONAL
COMP
```

Same as `BINARY`

```
COMPUTATIONAL-1
COMP-1
```

Same as `FLOAT-SHORT`

`COMPUTATIONAL-2`
`COMP-2`

Same as `FLOAT-LONG`

`COMPUTATIONAL-3`
`COMP-3`

Same as `PACKED-DECIMAL`

`COMPUTATIONAL-4`
`COMP-4`

Same as `BINARY`

`COMPUTATIONAL-5`
`COMP-5`

Range of Values:	Depends on number of '`9`'s in the `PICTURE` and the `binary-size` setting of the configuration file used to compile the program
Storage Format:	Native Binary Integer
Negative Values Allowed?:	If `PICTURE` contains '`S`'
`PICTURE` Used?:	Yes

`COMPUTATIONAL-6`
`COMP-6`

Range of Values:	Defined by the quantity of '`9`'s and the presence or absence of an '`S`' in the `PICTURE`
Storage Format:	Unsigned Packed Decimal
Negative Values Allowed?:	No
`PICTURE` Used?:	Yes

`COMPUTATIONAL-X`
`COMP-X`

Range of Values:	If used with `PIC X`, allocates one byte of storage per '`X`'; range of values is 0 to max storable in that many bytes. If used with `PIC 9`, range of values depends on number of '`9`'s in PICTURE

Storage Format: Native unsigned (X) or signed (9) Binary

Negative Values Allowed?: If `PICTURE` 9 and contains '`S`'

`PICTURE` Used?: Yes

`DISPLAY`
~~~~~~~

Range of Values: Depends on `PICTURE` — One character per X, A, 9, period, $, Z, 0, *, S (if `SEPARATE CHARACTER` specified), `+`, - or B symbol in `PICTURE`; Add 2 more bytes if the `DB` or `CR` editing symbol is used

Storage Format: Characters

Negative Values Allowed?: If `PICTURE` contains '`S`'

`PICTURE` Used?: Yes

`FLOAT-DECIMAL-16`
~~~~~~~~~~~~~~~~

Range of Values: -9.999999999999999 * 10^{383} – 9.999999999999999 * 10^{384}

Storage Format: Native IEEE 754 Decimal64 Floating-point

Negative Values Allowed?: Yes

`PICTURE` Used?: No

`FLOAT-DECIMAL-34`
~~~~~~~~~~~~~~~~

Range of Values: -9.99999... * 10^{6143} – 9.99999... * 10^{6144}

Storage Format: Native IEEE 754 Decimal128 Floating-point

Negative Values Allowed?: Yes

`PICTURE` Used?: No

`FLOAT-LONG`
~~~~~~~~~~

Range of Values: Approximately -1.797693134862316 * 10^{308} – 1.797693134862316 * 10^{308}

Storage Format: Native IEEE 754 Binary64 Floating-point

Negative Values Allowed?: Yes

`PICTURE` Used?: No

`FLOAT-SHORT`
~~~~~~~~~~~~

Range of Values: Approximately -3.4028235 * 10^{38} – 3.4028235 * 10^{38}

Storage Format: Native IEEE 754 Binary32

Negative Values Allowed?: Yes

`PICTURE` Used?: No

`INDEX`
~~~~~

Range of Values: 0 to maximum address possible (32 or 64 bits)

Storage Format: Native Binary Integer

Negative Values Allowed?: No

`PICTURE` Used?: No

`NATIONAL`
~~~~~~~~

`USAGE NATIONAL`, while syntactically recognized, is not supported by GnuCOBOL

`PACKED-DECIMAL`
~~~~~~~~~~~~~~

Range of Values: Defined by the quantity of '`9`'s and the presence or absence of an '`S`' in the PICTURE

Storage Format: Signed Packed Decimal

Negative Values Allowed?: If `PICTURE` contains '`S`'

`PICTURE` Used?: Yes

`POINTER`
~~~~~~~
~~~~~~~

Range of Values: 0 to maximum address possible (32 or 64 bits)

Storage Format: Native Binary Integer

Negative Values Allowed?: No

`PICTURE` Used?: No

`PROCEDURE-POINTER`

Same as `PROGRAM-POINTER`

`PROGRAM-POINTER`

Range of Values: 0 to maximum address possible (32 or 64 bits)

Storage Format: Native Binary Integer

Negative Values Allowed?: No

`PICTURE` Used?: No

`SIGNED-INT`

Same as `BINARY-LONG SIGNED`

`SIGNED-LONG`

Same as `BINARY-DOUBLE SIGNED`

`SIGNED-SHORT`

Same as `BINARY-SHORT SIGNED`

`UNSIGNED-INT`

Same as `BINARY-LONG UNSIGNED`

`UNSIGNED-LONG`

Same as `BINARY-DOUBLE UNSIGNED`

`UNSIGNED-SHORT`

Same as `BINARY-SHORT UNSIGNED`

5. `USAGE IS type-name-1`
   `~~~~~ ~~`

6. This USAGE clause indicates that the data description of the subject of the entry is specified by a user-defined data type. The user-defined data type is defined using the TYPEDEF clause, which is described in TYPEDEF clause.

```
Example 1:

01 struct-1     TYPEDEF.
    05  part-1  pic x(20).
    05  part-2  pic x(10).
01  a.
    05  b  USAGE struct-1.
    05  x  USAGE USHORT.
01 USHORT      pic 9(04) comp-5 typedef.

Which would be interpreted as if it had been coded as:

 01  a.
     05  b.
         10  part-1 pic x(20).
         10  part-2 pic x(10).
     05  x          pic 9(04) comp-5.

Example 2:

   IDENTIFICATION   DIVISION.
   PROGRAM-ID.      prog.
   DATA             DIVISION.
   WORKING-STORAGE  SECTION.
   01 MAIN-FILE-NAME-T     PIC X(50) IS TYPEDEF.
   01 DETAIL-NO-T             PIC 9999  IS TYPEDEF.
   01 MAIN-FILE-NAME-2T              IS TYPEDEF.
      05 FILLER               PIC 9999.
      05 DETAIL-NO            USAGE DETAIL-NO-T.

   01  MESSAGE-TEXT-2T        IS TYPEDEF.
     02 MAIN-FILE-NAME     USAGE MAIN-FILE-NAME-T.
     02 FILLER                REDEFINES MAIN-FILE-NAME.
         05 FILLER            PIC 9999.
     02 MAIN-FILE-NAME-2   USAGE MAIN-FILE-NAME-2T.
     02 FILLER                USAGE MAIN-FILE-NAME-T.
```

```
01  MESSAGE-TEXT-2 EXTERNAL USAGE MESSAGE-TEXT-2T.

77 OUTPUT-NAME              USAGE DETAIL-NO-T GLOBAL.

01 Z-MESSAGE-T2             USAGE MAIN-FILE-NAME-2T.
01 Z-MESSAGE-T3.
   49 MT3                   USAGE MESSAGE-TEXT-2T.
   49 MT3-REN REDEFINES MT3 USAGE MESSAGE-TEXT-2T.

PROCEDURE DIVISION.
    DISPLAY MAIN-FILE-NAME OF MESSAGE-TEXT-2
    DISPLAY DETAIL-NO         OF Z-MESSAGE-T2
    DISPLAY MAIN-FILE-NAME OF MT3
    DISPLAY OUTPUT-NAME
    GOBACK.
```

Example 3:

```
77  INT                     PIC S9(09)COMP-5 IS TYPEDEF.
01  Z-RETURNCODE            USAGE INT VALUE 0.
01  MESSAGE-TEXT-2                           IS TYPEDEF.
    02 MAIN-FILE-NAME    PIC X(50).
    02 FILLER               PIC X(79).
01 Z-MESSAGE-T2 USAGE MESSAGE-TEXT-2.
77 VIRMSG-TEXT              PIC  X(129) IS TYPEDEF.
01  MESSAGE-TEXT-2                        IS TYPEDEF.
    02 MAIN-FILE-NAME    PIC X(50).
    02 FILLER               PIC X(79).
01 Z-MESSAGE-TEXT           USAGE VIRMSG-TEXT.
01 Z-MESSAGE-T2 REDEFINES Z-MESSAGE-TEXT USAGE MESSAGE-TEXT-2.
01  W102-TYPE IS TYPEDEF.
    COPY SERVERFIELDS.
01 T-A                      USAGE W102-TYPE.
01 T-N                      USAGE W102-TYPE.
```

7. Binary data (integer or floating-point) can be stored in either a *Big-Endian* or *Little-Endian* form.

   Big-endian data allocation calls for the bytes that comprise a binary item to be allocated such that the least-significant byte is the right-most byte. For example, a four-byte binary item having a value of decimal 20 would be big-endian allocated as 00000014 (shown in hexadecimal notation).

   Little-endian data allocation calls for the bytes that comprise a binary item to be allocated such that the least-significant byte is the left-most byte. For example, a four-byte binary item having a value of decimal 20 would be little-endian allocated as 14000000 (shown in hexadecimal notation).

All CPUs are capable of *understanding* big-endian format, which makes it the *most compatible* form of binary storage across computer systems.

Some CPUs — such as the Intel/AMD i386/x64 architecture processors used in most Windows PCs — prefer to process binary data stored in a little-endian format. Since that format is more efficient on those systems, it is referred to as the *native* binary format.

On a system supporting only one format of binary storage (generally, that would be big-endian), the terms *most efficient* and *native* format are synonymous.

8. Data items that have the `UNSIGNED` attribute explicitly coded, or `DISPLAY`, `PACKED-DECIMAL`, `COMP-5`, `COMP-X` items that do not have an '`S`' symbol in their picture clause cannot preserve negative values that may be stored into them. Storing a negative value into such a field will actually result in the sign being stripped, essentially saving the absolute value in the data item.
9. Packed-decimal (*i.e.* `USAGE PACKED-DECIMAL`, `USAGE COMP-3` or `USAGE COMP-6`) data is stored as a series of bytes such that each byte contains two 4-bit fields, referred to as *nibbles* (since they comprise half a "byte", they're just "nibbles" — don't groan, I don't just make this stuff up!). Each nibble represents a '`9`' in the `PICTURE` and each holds a single decimal digit encoded as its binary value (0 = 0000, 1 = 0001, . . . , 9 = 1001).

   The *last* byte of a `PACKED-DECIMAL` or `COMP-3` data item will always have its left nibble corresponding to the last '`9`' in the `PICTURE` and its right nibble reserved as a sign indicator. This sign indicator is always present regardless of whether or not the `PICTURE` included an '`S`' symbol.

   The *first* byte of the data item will contain an unused left nibble if the `PICTURE` had an even number of '`9`' symbols in it.

   The sign indicator will have a value of a hexadecimal A through F. Traditional packed decimal encoding rules call for hexadecimal values of F, A, C or E ("FACE") in the sign nibble to indicate a positive value and B or D to represent a negative value (hexadecimal digits 0-9 are undefined). Testing with a Windows MinGW/GnuCOBOL implementation shows that — in fact — hex digit D represents a negative number and any other hexadecimal digit denotes a positive number. Therefore, a `PIC S9(3) COMP-3` packed-decimal field with a value of -15 would be stored internally as a hexadecimal 015D in GnuCOBOL.

   If you attempt to store a negative number into a packed decimal field that has no '`S`' in its `PICTURE`, the absolute value of the negative number will actually be stored.

   `USAGE COMP-6` does not allow for negative values, therefore no sign nibble will be allocated. A `USAGE COMP-6` data item containing an odd number of '`9`' symbols in its `PICTURE` will leave its leftmost nibble unused.
10. The `USAGE` specifications `FLOAT-DECIMAL-16` and `FLOAT-DECIMAL-34` will encode data using IEEE 754 *Decimal64* and *Decimal128* format, respectively. The former allows for up to 16 digits of exact precision while the latter offers 34. The phrase "exact precision" is used because the traditional binary renderings of decimal real numbers in a floating-point format (`FLOAT-LONG` and `FLOAT-SHORT`, for example) only yield an approximation of the actual value because many decimal fractions cannot be precisely

rendered in binary. The Decimal64 and Decimal128 renderings, however, render decimal real numbers in encoded decimal form in much the same way that `PACKED-DECIMAL` renders a decimal integer in digit-by-digit decimal form. The exact manner in which this rendering is performed is complex (Wikipedia has an excellent article on the subject — just search for *Decimal64*).

11. GnuCOBOL stores `FLOAT-DECIMAL-16` and `FLOAT-DECIMAL-34` data items using either Big-Endian or Little-Endian form, whichever is native to the system.
12. The `USAGE` specifications `FLOAT-LONG` and `FLOAT-SHORT` use the IEEE 754 *Binary64* and *Binary32* formats, respectively. These are binary encodings of real decimal numbers, and as such cannot represent every possible value between the minimum and maximum values in the range for those usages. Wikipedia has an excellent article on the Binary64 and Binary32 encoding schemes — just search on *Binary32* or *Binary64*.

    GnuCOBOL stores `FLOAT-LONG` and `FLOAT-SHORT` data items using either Big-Endian or Little-Endian form, whichever is native to the system.
13. A `USAGE` clause specified at the group item level will apply that `USAGE` to all subordinate data items, except those that themselves have a `USAGE` clause.
14. The only `USAGE` that is allowed in the report section is `USAGE DISPLAY`.

6.9.52. USING

USING Clause Syntax

```
USING identifier-1
~~~~~
```

This syntax is valid in the following sections: `SCREEN`

This clause logically attaches a screen section data item to another data item defined elsewhere in the data division.

1. When the screen item whose definition this clause is part of is displayed, the value currently in *identifier-1* will be automatically moved into the screen item first.
2. When the screen item whose definition this clause is part of (or its parent) is accepted, the current contents of the screen item will be saved back to *identifier-1* at the conclusion of the `ACCEPT`.
3. The `FROM` (see [FROM], page 174), `TO` (see [TO], page 221), `USING` and `VALUE` (see [VALUE], page 244) clauses are mutually-exclusive in any screen section data item's definition.

6.9.53. VALUE

VALUE (Condition Names) Clause Syntax

```
{ VALUE IS   } {literal-1 [ THRU|THROUGH literal-2 ]}...
{ ~~~~~      }              ~~~~ ~~~~~~~
{ VALUES ARE }
  ~~~~~~
    [ WHEN SET TO FALSE IS literal-3 ]
                  ~~~~~
```

VALUE (Other Data Items) Syntax

```
VALUE IS [ ALL ] literal-1
~~~~~      ~~~
```

This syntax is valid in the following sections: `FILE`, `WORKING-STORAGE`, `LOCAL-STORAGE`, `LINKAGE`, `REPORT`, `SCREEN`

The `VALUE` clause is used to define condition names or to assign values (at compilation time) to data items.

1. The reserved words `ARE` and `IS` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. This clause cannot be specified on the same data item as a `FROM` (see [FROM], page 174), `TO` (see [TO], page 221) or `USING` (see [USING], page 243) clause.
3. The following points apply to using the `VALUE` clause in the definition of a condition name:
   A. The clauses `VALUE IS` and `VALUES ARE` are interchangeable.
   B. The reserved words `THRU` and `THROUGH` are interchangeable.
   C. See [88-Level Data Items], page 153, for a discussion of how this format of `VALUE` is used to create condition names.
   D. See [Condition Names], page 45, for a discussion of how condition names are used.
4. The following points apply to using the `VALUE` clause in the definition of any other data item:
   A. In this context, `VALUE` specifies an initial compilation-time value that will be assigned to the storage occupied by the data item in the program object code generated by the compiler.
   B. The `VALUE` clause is ignored on `EXTERNAL` (see [EXTERNAL], page 171) data items or on any data items defines as subordinate to an `EXTERNAL` data item.
   C. This format of the `VALUE` clause may not be used anywhere in the description of an 01 item (or any of its subordinate items) serving as an `FD` or `SD` record description.

D. If the optional `ALL` clause is used, it may only be used with an alphanumeric literal value; the value will be repeated as needed to completely fill the data item. Here are some examples with and without `ALL` (the symbol *b* denotes a space):

```
PIC X(5) VALUE 'A'      *> Abbbb
PIC X(5) VALUE ALL 'A'  *> AAAAA
PIC 9(3) VALUE 1        *> 001
PIC 9(3) VALUE ALL '1'  *> 111
```

E. When used in the definition of a screen data item:

   a. A figurative constant may not be supplied as *literal-1*.

   b. Any `FROM` (see [FROM], page 174), `TO` (see [TO], page 221) or `USING` (see [USING], page 243) clause in the same data item's definition will be ignored.

   c. If there is no picture clause specified, the size of the screen data item will be the length of the *literal-1* value.

   d. If there is no picture clause and the `ALL` option is specified, the `ALL` option will be ignored.

F. Giving a table an initial, compile-time value is one of the trickier aspects of COBOL data definition. There are basically three standard techniques and a fourth that people familiar with other COBOL implementations but new to GnuCOBOL may find interesting. So, here are the three *standard* approaches:

   a. Don't bother worrying about it at compile-time. Use the `INITIALIZE` (see [INITIALIZE], page 339) to initialize all data item occurrences in a table (at run-time) to their data-type-specific default values (numerics: 0, alphabetic and alphanumerics: spaces).

   b. Initialize small tables at compile time by including a `VALUE` clause on the group item that serves as a parent to the table, as follows:

```
05  SHIRT-SIZES          VALUE "S 14M 15L 16XL17".
    10 SHIRT-SIZE-TBL    OCCURS 4 TIMES.
       15 SST-SIZE       PIC X(2).
       15 SST-NECK       PIC 9(2).
```

   c. Initialize tables of almost any size at compilation time by utilizing the `REDEFINES` (see [REDEFINES], page 204) clause:

```
05  SHIRT-SIZE-VALUES.
    10 PIC X(4)          VALUE "S 14".
    10 PIC X(4)          VALUE "M 15".
    10 PIC X(4)          VALUE "L 16".
    10 PIC X(4)          VALUE "XL17".
05  SHIRT-SIZES          REDEFINES SHIRT-SIZE-VALUES.
    10 SHIRT-SIZE-TBL    OCCURS 4 TIMES.
       15 SST-SIZE       PIC X(2).
       15 SST-NECK       PIC 9(2).
```

   Admittedly, this table is much more verbose than the one shown with a group `VALUE`. What is good about this initialization technique, however, is that you can have as many `FILLER` and `VALUE` items as you need for a larger table, and those values can be as long as necessary!

G. Many COBOL compilers do not allow the use of `VALUE` and `OCCURS` (see [OCCURS], page 190) on the same data item; additionally, they don't allow a `VALUE` clause on a data item subordinate to an `OCCURS`. GnuCOBOL, however, has neither of these restrictions!

   Observe the following example, which illustrates a fourth manner in which tables may be initialized in GnuCOBOL:

```
05  X           OCCURS 6 TIMES.
    10 A        PIC X(1) VALUE '?'.
    10 B        PIC X(1) VALUE '%'.
    10 N        PIC 9(2) VALUE 10.
```

   In this example, all six '`A`' items will be initialized to '`?`', all six '`B`' items will be initialized to '`%`' and all six '`N`' items will be initialized to 10. It's not clear exactly how many times this sort of initialization will be useful, but it's there if you need it.

5. The `FROM` (see [FROM], page 174), `TO` (see [TO], page 221), `USING` (see [USING], page 243) and `VALUE` clauses are mutually-exclusive in any screen section data item's definition.

End of Chapter 6 — DATA DIVISION

7. PROCEDURE DIVISION

PROCEDURE DIVISION Syntax

```
  PROCEDURE DIVISION [ { USING Subprogram-Argument ...        } ]
  ~~~~~~~~~ ~~~~~~~~   { ~~~~~                                }
                       { CHAINING Main-Program-Argument...}
                         ~~~~~~~~
                     [ RETURNING identifier-1 ] .
[ DECLARATIVES. ]      ~~~~~~~~~
  ~~~~~~~~~~~~
[ Event-Handler-Routine...  .  ]

[ END DECLARATIVES. ]
  ~~~ ~~~~~~~~~~~~
  General-Program-Logic

[ Nested-Subprogram...  ]

[ END PROGRAM|FUNCTION name-1 ]
  ~~~ ~~~~~~~ ~~~~~~~~
```

The `PROCEDURE DIVISION` of any GnuCOBOL program marks the point where all executable code is written.

7.1. PROCEDURE DIVISION USING

PROCEDURE DIVISION Subprogram-Argument Syntax

```
[ BY { REFERENCE [ OPTIONAL ]                          } ] identifier-1
     { ~~~~~~~~~   ~~~~~~~~                            }
     { VALUE [ [ UNSIGNED ] SIZE IS { AUTO      } ] }
       ~~~~~     ~~~~~~~~   ~~~~    { ~~~~      }
                                    { DEFAULT   }
                                    { ~~~~~~~   }
                                    { integer-1 }
```

The `USING` clause defines the arguments that will be passed to a GnuCOBOL program which is serving as a subprogram.

1. The reserved words `BY` and `IS` are optional and may be omitted. The presence or absence of these words have no effect upon the program.
2. The `USING` clause should only be used on the procedure division header of subprograms (subroutines or user-defined functions).
3. The calling program will pass zero or more data items, known as arguments, to this subprogram — there must be exactly as many *identifier-1* data items specified on the `USING` clause as the maximum number of arguments the subprogram will ever be passed.
4. If a subprogram does not expect any arguments, it should not have a `USING` clause specified on its procedure division header.
5. The order in which arguments are defined on the `USING` clause must correspond to the order in which those arguments will be passed to the subprogram by the calling program.
6. The identifiers specified on the `USING` clause must be defined in the linkage section of the subprogram. No storage is actually allocated for those identifiers in the subprogram as the actual storage for them will exist in the calling program.
7. A GnuCOBOL subprogram expects that all arguments to it will be one of two things:
   - The memory address of the actual data item (allocated in the calling program) that is being passed to the subprogram.
   - A numeric, full-word, binary value (i.e. `USAGE BINARY-LONG` (see [USAGE], page 232)) which is the actual argument being passed to the subprogram.

   In the case of the former, the `USING` clause on the procedure division header should describe the argument via the `BY REFERENCE` clause — in the latter case, a `BY VALUE` specification should be coded. This allows the code generated by the compiler to properly reference the subprogram arguments at run-time.
8. `BY REFERENCE` is the assumed default for the first `USING` argument should no `BY` clause be specified for it. Subsequent arguments will assume the `BY` specification of the argument prior to them should they lack a `BY` clause of their own.

9. Changes made by a subprogram to the value of an argument specified on the `USING` clause will "be visible" to the calling program only if `BY REFERENCE` was explicitly specified or implicitly assumed for the argument on the subprogram's procedure division header *and* the argument was passed to the subprogram `BY REFERENCE` by the calling program. See [Subprogram Arguments], page 686, for additional information on the mechanics of how arguments are passed to subprograms.
10. The optional `SIZE` clause allows you to specify the number of bytes a `BY VALUE` argument will occupy, with `SIZE DEFAULT` specifying 4 bytes (this is the default if no `SIZE` clause is used), `SIZE AUTO` specifying the size of the argument in the calling program and `SIZE` *`integer-1`* specifying a specific byte count.
11. The optional `UNSIGNED` keyword, legal only if `SIZE AUTO` or `SIZE` *`integer-1`* are coded, will add the `unsigned` attribute to the argument's specification in the C-language function header code generated for the subprogram. While not of any benefit when the calling program is a GnuCOBOL program, this can improve compatibility with a C-language calling program.
12. The `OPTIONAL` keyword, legal only on `BY REFERENCE` arguments, allows calling programs to code `OMITTED` for that corresponding argument when they call this subprogram. See [CALL], page 293. for additional information on this feature.

7.2. PROCEDURE DIVISION CHAINING

PROCEDURE DIVISION Main-Program-Argument Syntax

```
[ BY REFERENCE ] [ OPTIONAL ] identifier-1
     ~~~~~~~~~     ~~~~~~~~
```

The CHAINING term provides one mechanism a programmer may use to retrieve command-line arguments passed to a program at execution time.

1. PROCEDURE DIVISION CHAINING may only be coded in a main program (that is, the first program executed when a compiled GnuCOBOL compilation unit is executed). It cannot be used in any form of subprogram.
2. The CHAINING clause defines arguments that will be passed to a main program from the operating system. The argument identifiers specified on the CHAINING clause will be populated by character strings comprised of the parameters specified to the program on the command line that executed it, as follows:
   A. When a GnuCOBOL program is executed from a command-line, the complete command line text will be broken into a series of *tokens*, where each token is identified as being a word separated from the others in the command text by at least one space. For example, if the command line was `/usr/local/myprog THIS IS A TEST`, there will be five tokens identified by the operating system — '`/usr/local/myprog`', '`THIS`', '`IS`', '`A`' and '`TEST`'.
   B. Multiple space-delimited tokens may be treated as a single token by enclosing them in quotes. For example, there are only three tokens generated from the command line `C:\Pgms\myprog.exe 'THIS IS A' TEST` — '`C:\Pgms\myprog.exe`', '`THIS IS A`' and '`TEST`'. When quote characters are used to create multi-word tokens, the quote characters themselves are stripped from the token's value.
   C. Once tokens have been identified, the first one (the command) will be discarded; the rest will be stored into the CHAINING arguments when the program begins execution, with the 2nd token going to the 1st argument, the 3rd token going to the 2nd argument and so forth.
   D. If there are more tokens than there are arguments, the excess tokens will be discarded.
   E. If there are fewer tokens than there are arguments, the excess arguments will be initialized as if the INITIALIZE *identifier-1* (see [INITIALIZE], page 339) statement were executed.
   F. All identifiers specified on the CHAINING clause should be defined as PIC X, PIC A, group items (which are treated implicitly as PIC X) or as PIC 9 USAGE DISPLAY. The use of USAGE BINARY (or the like) data items as CHAINING arguments is not recommended as all command-line tokens will be retained in their original character form as they are moved into the argument data items.
   G. If an argument identifier is smaller in storage size than the token value to be stored in it, the right-most excess characters of the token value will be truncated as the

value is moved in. Any `JUSTIFIED RIGHT` clause on such an argument identifier will be ignored.

H. If an argument is larger in storage size than the token value to be stored in it, the token value will be moved into the argument identifier in a left-justified manner. unmodified-modified byte positions in the identifier will be space filled, unless the argument identifier is defined as `PIC 9 USAGE DISPLAY`, in which case unmodified bytes will be filled with '`0`' characters from the systems native character set.

   This behaviour when the argument is defined as `PIC 9` may be unacceptable, as an argument defined as `PIC 9(3)` but passed in a value of '`1`' from the command line will receive a value of '`100`', not '`001`'. Consider defining "numeric" command line arguments as `PIC X` and then using the `NUMVAL` intrinsic function (see [NUMVAL], page 496) function to determine the proper numeric value.

7.3. PROCEDURE DIVISION RETURNING

PROCEDURE DIVISION RETURNING Syntax

```
RETURNING identifier-1
~~~~~~~~~
```

The RETURNING clause on the PROCEDURE DIVISION header documents that the subprogram in which the clause appears will be returning a numeric value back to the program that called it.

1. The `RETURNING` clause is optional within a subroutine, as not all subroutines return a value to their caller.
2. The `RETURNING` clause is mandatory within a user-defined function, as all such must return a numeric result.
3. The *identifier-1* data item should be defined as a `USAGE BINARY-LONG` data item.
4. Main programs that wish to "pass back" a return code value to the operating system when they exit do not use `RETURNING` - they do so simply by MOVEing a value to the `RETURN-CODE` special register.
5. This is not the only mechanism that a subprogram may use to pass a value back to its caller. Other possibilities are:
   A. The subprogram may modify any argument that is specified as `BY REFERENCE` on its `PROCEDURE DIVISION` header. Whether the calling program can actually "see" any modifications depends upon how the calling program passed the argument to the subprogram. See [CALL], page 293, for more information.
   B. A data item with the `GLOBAL` (see [GLOBAL], page 176) attribute specified in its description in the calling program is automatically visible to and updatable by a subprogram nested with the calling program. See [Independent vs Contained vs Nested Subprograms], page 679, for more information on subprogram nesting.
   C. A data item defined with the `EXTERNAL` (see [EXTERNAL], page 171) attribute in a subprogram and the calling program (same name in both programs) is automatically visible to and updatable by both programs, even if those programs are compiled separately from one another.

7.4. PROCEDURE DIVISION Sections and Paragraphs

The procedure division is the only one of the COBOL divisions that allows you to create your own sections and paragraphs. These are collectively referred to as *Procedures*, and the names you create for those sections and paragraphs are called *Procedure Names*.

Procedure names are optional in the procedure division and — when used — are named entirely according to the needs and whims of the programmer.

Procedure names may be up to thirty one (31) characters long and may consist of letters, numbers, dashes and underscores. A procedure name may neither begin nor end with a dash ('-') or underscore ('_') character. This means that `Main`, `0100-Read-Transaction` and `17` are all perfectly valid procedure names.

There are three circumstances under which the use of certain GnuCOBOL statements or options will require the specification of procedures. These situations are:

1. When `DECLARATIVES` (see [DECLARATIVES], page 254) are specified.
2. When the `ENTRY` statement (see [ENTRY], page 318) is being used.
3. When any procedure division statement that references procedures is used. These statements are:
   - `ALTER` *`procedure-name`*
   - `GO TO` *`procedure-name`*
   - `MERGE ... OUTPUT PROCEDURE` *`procedure-name`*
   - `PERFORM` *`procedure-name`*
   - `SORT ... INPUT PROCEDURE` *`procedure-name`* and/or `SORT ... INPUT PROCEDURE` *`procedure-name`*

7.5. DECLARATIVES

DECLARATIVES Syntax

```
section-name-1 SECTION.

 USE { [ GLOBAL ] AFTER STANDARD { EXCEPTION } PROCEDURE ON { INPUT       } }
 ~~~ {   ~~~~~~                  { ~~~~~~~~~ }             { ~~~~~       } }
     {                           { ERROR     }             { OUTPUT      } }
     {                             ~~~~~                   { ~~~~~~      } }
     {                                                     { I-O         } }
     { FOR DEBUGGING ON { procedure-name-1          }      { ~~~         } }
     {     ~~~~~~~~~    { ALL PROCEDURES            }      { EXTEND      } }
     {                  { ~~~ ~~~~~~~~~~            }      { ~~~~~~      } }
     {                  { REFERENCES OF identifier-1 }      { file-name-1 } }
     {                                                                     }
     { [ GLOBAL ] BEFORE REPORTING identifier-2                            }
     {   ~~~~~~   ~~~~~~ ~~~~~~~~~                                         }
     {                                                                     }
     { AFTER EC|{EXCEPTION CONDITION}                                      }
             ~~  ~~~~~~~~~ ~~~~~~~~~
```

The `AFTER EXCEPTION CONDITION` and `AFTER EC` clauses are syntactically recognized but are otherwise non-functional.

The `DECLARATIVES` area of the procedure division allows the programmer to define a series of "trap" procedures (referred to as declarative procedures) capable of intercepting certain events that may occur at program execution time. The syntax diagram above shows the format of a single such procedure.

1. The reserved words `AFTER`, `FOR`, `ON`, `PROCEDURE` and `STANDARD` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. `EC` and `EXCEPTION CONDITION` are interchangeable.
3. The declaratives area may contain any number of declarative procedures, but no two declarative procedures should be coded to trap the same event.
4. The following points apply to the `USE BEFORE REPORTING` clause:
   A. *identifier-2* must be a report group.
   B. At run-time, the declaratives procedure will be executed prior to the processing of the specified report group's presentation; within the procedure you may take either of the following actions:
      - You may adjust the value(s) of any items referenced in `SUM` (see [SUM], page 520) or `SOURCE` (see [SOURCE], page 215) clauses in the report group.
      - You may execute the `SUPPRESS` (see [SUPPRESS], page 409) statement to squelch the presentation of the specified report group altogether. Note that

you will be suppressing this one specific instance of that group's presentation and not all of them.

5. The following points apply to the `USE FOR DEBUGGING` clause:

   A. This clause allows you to define a declarative procedure that will be invoked whenever. . .

      - . . . *identifier-1* is referenced on any statement.
      - . . . *procedure-name-1* is executed.
      - . . . any procedure is executed (`ALL PROCEDURES`).

   B. A `USE FOR DEBUGGING` declarative procedure will be ignored at *compilation* time unless `WITH DEBUGGING MODE` is specified in the `SOURCE-COMPUTER` (see [SOURCE-COMPUTER], page 89) paragraph. Neither the compiler's `-fdebugging-line` switch nor `-debug` switch will activate this feature.

   C. Any `USE FOR DEBUGGING` declarative procedures will be ignored at *execution* time unless the `COB_SET_DEBUG` run-time environment variable (see [Run Time Environment Variables], page 658) has been set to a value of '`Y`', '`y`' or '`1`'.

   D. The typical use of a `USE FOR DEBUGGING` declarative procedure is to display the `DEBUG-ITEM` special register, which will be implicitly and automatically created in your program for you if `WITH DEBUGGING MODE` is active.

      The structure of `DEBUG-ITEM` will be as follows:

```
01  DEBUG-ITEM.
    05 DEBUG-LINE      PIC X(6).
    05 FILLER          PIC X(1) VALUE SPACE.
    05 DEBUG-NAME      PIC X(31).
    05 FILLER          PIC X(1) VALUE SPACE.
    05 DEBUG-SUB-1     PIC S9(4) SIGN LEADING SEPARATE.
    05 FILLER          PIC X(1) VALUE SPACE.
    05 DEBUG-SUB-2     PIC S9(4) SIGN LEADING SEPARATE.
    05 FILLER          PIC X(1) VALUE SPACE.
    05 DEBUG-SUB-3     PIC S9(4) SIGN LEADING SEPARATE.
    05 FILLER          PIC X(1) VALUE SPACE.
    05 DEBUG-CONTENTS  PIC X(31).
```

      where. . .

      `DEBUG-LINE`
      : . . . is the program line number of the statement that triggered the declaratives procedure.

      `DEBUG-NAME`
      : . . . is the procedure name or identifier name that triggered the declaratives procedure.

      `DEBUG-SUB-1`
      : . . . is the first subscript value (if any) for the reference of the identifier that triggered the declaratives procedure.

`DEBUG-SUB-2`
: . . . is the second subscript value (if any) for the reference of the identifier that triggered the declaratives procedure.

`DEBUG-SUB-3`
: . . . is the third subscript value (if any) for the reference of the identifier that triggered the declaratives procedure.

`DEBUG-CONTENTS`
: . . . is a (brief) statement of the manner in which the procedure that triggered the declaratives procedure was executed or the first 31 characters of the value of the identifier whose reference triggered the declaratives procedure (the value after the statement was executed).

6. The `USE AFTER STANDARD ERROR PROCEDURE` clause defines a declarative procedure invoked any time a failure is encountered with the specified I/O type (or against the specified file(s)).
7. The `GLOBAL` (see [GLOBAL], page 176) option, if used, allows a declarative procedure to be used across the program containing the `USE` statement and any subprograms nested within that program.
8. Declarative procedures may not reference any other procedures defined outside the scope of DECLARATIVES.

7.6. Common Clauses on Executable Statements

7.6.1. AT END + NOT AT END

AT END Syntax

```
[ AT END imperative-statement-1 ]
     ~~~
[ NOT AT END imperative-statement-2 ]
  ~~~    ~~~
```

`AT END` clauses may be specified on `READ` (see [READ], page 366), `RETURN` (see [RETURN], page 373), `SEARCH` (see [SEARCH], page 377) and `SEARCH ALL` (see [SEARCH ALL], page 379) statements.

1. The following points pertain to the use of these clauses on `READ` (see [READ], page 366) and `RETURN` (see [RETURN], page 373) statements:
   A. The `AT END` clause will — if present — cause *imperative-statement-1* (see [Imperative Statement], page 722) to be executed if the statement fails due to a file status of 10 (end-of-file). See [File Status Codes], page 107, for a list of possible File Status codes.

      An `AT END` clause *will not detect other non-zero file-status values.*

Use a `DECLARATIVES` (see [DECLARATIVES], page 254) routine or an explicitly-declared file status field tested after the `READ` or `RETURN` to detect error conditions other than end-of-file.

B. A `NOT AT END` clause will cause *imperative-statement-2* to be executed if the `READ` or `RETURN` attempt is successful.

2. The following points pertain to the use of these clauses on `SEARCH` (see [SEARCH], page 377) and `SEARCH ALL` (see [SEARCH ALL], page 379) statements:

   A. An `AT END` clause detects and handles the case where either form of table search has failed to locate an entry that satisfies the search conditions being used.

   B. The `NOT AT END` clause is not allowed on either form of table search.

7.6.2. CORRESPONDING

Three GnuCOBOL statements — `ADD` (see [ADD CORRESPONDING], page 288), `MOVE` (see [MOVE CORRESPONDING], page 353) and `SUBTRACT` (see [SUBTRACT CORRESPONDING], page 407) support the use of a `CORRESPONDING` option:

```
ADD CORRESPONDING group-item-1 TO group-item-2
MOVE CORRESPONDING group-item-1 TO group-item-2
SUBTRACT CORRESPONDING group-item-1 FROM group-item-2
```

This option allows one or more data items within one group item (*group-item-1* — the first named on the statement) to be paired with correspondingly-named (hence the name) in a second group item (*group-item-2* — the second named on the statement). The contents of *group-item-1* will remain unaffected by the statement while one or more data items within *group-item-2* will be changed.

In order for *data-item-1*, defined subordinate to group item *group-item-1* to be a *corresponding* match to *data-item-2* which is subordinate to *group-item-2*, each of the following must be true:

1. Both *data-item-1* and *data-item-2* must have the same name, and that name may not explicitly or implicitly be `FILLER`.
2. Both *data-item-1* and *data-item-2*...
   A. ...must exist at the same relative structural "depth" of definition within *group-item-1* and *group-item-2*, respectively
   B. ...and all "parent" data items defined within each group item must have identical (but non-`FILLER`) names.
3. When used with a `MOVE` verb...
   A. ...one of *data-item-1* or *data-item-2* (but not both) is allowed to be a group item
   B. ...and it must be valid to move *data-item-1* TO *data-item-2*.
4. When used with `ADD` or `SUBTRACT` verbs, both *data-item-1* and *data-item-2* must be numeric, elementary, unedited items.
5. Neither *data-item-1* nor *data-item-2* may be a `REDEFINES` (see [REDEFINES], page 204) or `RENAMES` (see [RENAMES], page 205) of another data item.
6. Neither *data-item-1* nor *data-item-2* may have an `OCCURS` (see [OCCURS], page 190) clause, although either may contain subordinate data items that *do* have an `OCCURS` clause (assuming rule 3a applies)

Observe the definitions of data items '`Q`' and '`Y`'...

```
01  Q.                          01  Y.
    03 X.                            02 A         PIC X(1).
       05 A         PIC 9(1).        02 G1.
       05 G1.                           03 G2.
          10 G2.                           04 B   PIC X(1).
             15 B   PIC X(1).         02 C         PIC X(1).
       05 C.                          02 G3.
          10 FILLER PIC X(1).              03 G5.
       05 G3.                                 04 D   PIC X(1).
```

```
   10 G4.                          03 G6     PIC X(1).
      15 D   PIC X(1).          02 E         PIC 9(1).
05 E         PIC X(1).          02 F         PIC X(1).
05 F         REDEFINES V1       02 G         PIC X(4).
             PIC X(1).          02 H         OCCURS 4 TIMES
05 G.                                        PIC X(1).
   10 G6     OCCURS 4 TIMES     66 I         RENAMES E.
             PIC X(1).          02 J.
05 H         PIC X(4).             03 K.
05 I         PIC 9(1).                04 L.
05 J.                                    05 M.
   10 K.
      15 M   PIC X(1).
```

The following are the valid CORRESPONDING matches, assuming the statement MOVE CORRESPONDING X TO Y is being executed (there are no valid corresponding matches for ADD CORRESPONDING or SUBTRACT CORRESPONDING because every potential match up violates rule #4):

A, B, C, G

The following are the CORRESPONDING match ups that passed rule #1 (but failed on another rule), and the reasons why they failed.

Data Item	Failure Reason
D	Fails due to rule #2b
E	Fails due to rule #3b
F	Fails due to rule #5
G1	Fails due to rule #3a
G2	Fails due to rule #3a
G3	Fails due to rule #3a
G4	Fails due to rule #1
G5	Fails due to rule #1
G6	Fails due to rule #6
H	Fails due to rule #6
I	Fails due to rule #5
J	Fails due to rule #3a
K	Fails due to rule #3a
L	Fails due to rule #1
M	Fails due to rule #2a

7.6.3. INVALID KEY + NOT INVALID KEY

INVALID KEY Syntax

```
[ INVALID KEY imperative-statement-1 ]
  ~~~~~~~
```

```
[ NOT INVALID KEY imperative-statement-2 ]
  ~~~ ~~~~~~~
```

`INVALID KEY` clauses may be specified on `DELETE` (see [DELETE], page 304), `READ` (see [Random READ], page 368), `REWRITE` (see [REWRITE], page 374), `START` (see [START], page 397) and `WRITE` (see [WRITE], page 417) statements.

Specification of an `INVALID KEY` clause will allow your program to trap an I/O failure condition (with an I/O error code in the file's `FILE-STATUS` (see [SELECT], page 104) field) that has occurred due to a record-not-found condition and handle it gracefully by executing *imperative-statement-1* (see [Imperative Statement], page 722).

An optional `NOT INVALID KEY` clause will cause *imperative-statement-2* to be executed if the statement's execution was successful.

7.6.4. ON EXCEPTION + NOT ON EXCEPTION

ON EXCEPTION Syntax

```
[ ON EXCEPTION imperative-statement-1 ]
    ~~~~~~~~~
[ NOT ON EXCEPTION imperative-statement-2 ]
  ~~~   ~~~~~~~~~
```

`EXCEPTION` clauses may be specified on `ACCEPT` (see [ACCEPT], page 268), `CALL` (see [CALL], page 293) and `DISPLAY` (see [DISPLAY], page 305) statements.

Specification of an exception clause will allow your program to trap a failure condition that has occurred and handle it gracefully by executing *imperative-statement-1* (see [Imperative Statement], page 722). If such a condition occurs at runtime without having one of these clauses specified, an error message will be generated (by the GnuCOBOL runtime library) to the `SYSERR` device (pipe 2). The program may also be terminated, depending upon the type and severity of the error.

An optional `NOT ON EXCEPTION` clause will cause *imperative-statement-2* to be executed if the statement's execution was successful.

7.6.5. ON OVERFLOW + NOT ON OVERFLOW

ON OVERFLOW Syntax

```
[ ON OVERFLOW imperative-statement-1 ]
    ~~~~~~~~
[ NOT ON OVERFLOW imperative-statement-2 ]
  ~~~   ~~~~~~~~
```

`OVERFLOW` clauses may be specified on `CALL` (see [CALL], page 293), `STRING` (see [STRING], page 401) and `UNSTRING` (see [UNSTRING], page 413) statements.

An `ON OVERFLOW` clause will allow your program to trap a failure condition that has occurred and handle it gracefully by executing *imperative-statement-1* (see [Imperative Statement], page 722). If such a condition occurs at runtime without having one of these clauses specified, an error message will be generated (by the GnuCOBOL runtime library) to the `SYSERR` device (pipe 2). The program may also be terminated, depending upon the type and severity of the error.

An optional `NOT ON OVERFLOW` clause will cause *imperative-statement-2* to be executed if the statement's execution was successful.

7.6.6. ON SIZE ERROR + NOT ON SIZE ERROR

ON SIZE ERROR Syntax

```
[ ON SIZE ERROR imperative-statement-1 ]
    ~~~~ ~~~~~
[ NOT ON SIZE ERROR imperative-statement-2 ]
  ~~~    ~~~~ ~~~~~
```

`SIZE ERROR` clauses may be included on `ADD` (see [ADD], page 284), `COMPUTE` (see [COMPUTE], page 301), `DIVIDE` (see [DIVIDE], page 312), `MULTIPLY` (see [MULTIPLY], page 354) and `SUBTRACT` (see [SUBTRACT], page 403) statements.

Including an `ON SIZE ERROR` clause on an arithmetic statement will allow your program to trap a failure of an arithmetic statement (either generating a result too large for the receiving field, or attempting to divide by zero) and handle it gracefully by executing *imperative-statement-1* (see [Imperative Statement], page 722). Field size overflow conditions occur silently, usually without any runtime messages being generated, even though such events rarely lend themselves to generating correct results. Division by zero errors, when no `ON SIZE ERROR` clause exists, will produce an error message (by the GnuCOBOL runtime library) to the `SYSERR` device (pipe 2) and will also abort the program.

An optional `NOT ON SIZE ERROR` clause will cause *imperative-statement-2* to be executed if the arithmetic statement's execution was successful.

7.6.7. ROUNDED

ROUNDED Syntax

```
ROUNDED [ MODE IS { AWAY-FROM-ZERO         }
~~~~~~~   ~~~~    { ~~~~~~~~~~~~~~         }
                  { NEAREST-AWAY-FROM-ZERO }
```

```
{ ~~~~~~~~~~~~~~~~~~~~~ }
{ NEAREST-EVEN          }
{ ~~~~~~~~~~~~          }
{ NEAREST-TOWARD-ZERO   }
{ ~~~~~~~~~~~~~~~~~~    }
{ PROHIBITED            }
{ ~~~~~~~~~~            }
{ TOWARD-GREATER        }
{ ~~~~~~~~~~~~~~        }
{ TOWARD-LESSER         }
{ ~~~~~~~~~~~~~         }
{ TRUNCATION            }
  ~~~~~~~~~~
```

GnuCOBOL provides for control over the final rounding process applied to the receiving fields on all arithmetic verbs. Each of the arithmetic statements (`ADD` (see [ADD], page 284), `COMPUTE` (see [COMPUTE], page 301), `DIVIDE` (see [DIVIDE], page 312), `MULTIPLY` (see [MULTIPLY], page 354) and `SUBTRACT` (see [SUBTRACT], page 403)) statements allow an optional `ROUNDED` clause to be applied to each receiving data item.

The following rules apply to the rounding behaviour induced by this clause.

1. Rounding only applies when the result being saved to a receiving field with a `ROUNDED` clause is a non-integer value.
2. Absence of a `ROUNDED` clause is the same as specifying `ROUNDED MODE IS TRUNCATION`.
3. Use of a `ROUNDED` clause without a `MODE` specification is the same as specifying `ROUNDED MODE IS NEAREST-AWAY-FROM-ZERO`.

The behaviour of the eight different rounding modes is defined in the following table. Note that a '`...`' indicates the last digit repeats. The examples assume an integer receiving field.

`AWAY-FROM-ZERO`

Rounding is to the nearest value of larger magnitude.

-3.510 ⇒ -4	+3.510 ⇒ +4
-3.500 ⇒ -4	+3.500 ⇒ +4
-3.499... ⇒ -4	+3.499... ⇒ +4
-2.500 ⇒ -3	+2.500 ⇒ +3
-2.499... ⇒ -3	+2.499... ⇒ +3

`NEAREST-AWAY-FROM-ZERO`

Rounding is to the nearest value (larger or smaller). If two values are equally near, the value with the larger absolute value is selected.

-3.510 ⇒ -4	+3.510 ⇒ +4
-3.500 ⇒ -4	+3.500 ⇒ +4
-3.499... ⇒ -3	+3.499... ⇒ +3
-2.500 ⇒ -3	+2.500 ⇒ +3

-2.499... ⇒ -2 +2.499... ⇒ +2

`NEAREST-EVEN`

Rounding is to the nearest value (larger or smaller). If two values are equally near, the value whose rightmost digit is *even* is selected. This mode is sometimes called "Banker's rounding".

-3.510 ⇒ -4 +3.510 ⇒ +4
-3.500 ⇒ -4 +3.500 ⇒ +4
-3.499... ⇒ -3 +3.499... ⇒ +3
-2.500 ⇒ -2 +2.500 ⇒ +2
-2.499... ⇒ -2 +2.499... ⇒ +2

`NEAREST-TOWARD-ZERO`

Rounding is to the nearest value (larger or smaller). If two values are equally near, the value with the smaller absolute value is selected.

-3.510 ⇒ -4 +3.510 ⇒ +4
-3.500 ⇒ -3 +3.500 ⇒ +3
-3.499... ⇒ -3 +3.499... ⇒ +3
-2.500 ⇒ -2 +2.500 ⇒ +2
-2.499... ⇒ -2 +2.499... ⇒ +2

`PROHIBITED`

No rounding is performed. If the value cannot be represented exactly in the desired format, the `EC-SIZE-TRUNCATION` condition (exception code 1005) is set (and may be retrieved via the `ACCEPT` (see [ACCEPT FROM Runtime-Info], page 281) statement) and the results of the operation are undefined.

-3.510 ⇒ Undefined +3.510 ⇒ Undefined
-3.500 ⇒ Undefined +3.500 ⇒ Undefined
-3.499... ⇒ Undefined +3.499... ⇒ Undefined
-2.500 ⇒ Undefined +2.500 ⇒ Undefined
-2.499... ⇒ Undefined +2.499... ⇒ Undefined

`TOWARD-GREATER`

Rounding is toward the nearest value whose algebraic value is larger.

-3.510 ⇒ -3 +3.510 ⇒ +4
-3.500 ⇒ -3 +3.500 ⇒ +4
-3.499... ⇒ -3 +3.499... ⇒ +4
-2.500 ⇒ -2 +2.500 ⇒ +3
-2.499... ⇒ -2 +2.499... ⇒ +3

`TOWARD-LESSER`

Rounding is toward the nearest value whose algebraic value is smaller.

-3.510 ⇒ -4 +3.510 ⇒ +3
-3.500 ⇒ -4 +3.500 ⇒ +3
-3.499... ⇒ -4 +3.499... ⇒ +3
-2.500 ⇒ -3 +2.500 ⇒ +2
-2.499... ⇒ -3 +2.499... ⇒ +2

`TRUNCATION`

Rounding is to the nearest value whose magnitude is smaller.

-3.510 ⇒ -3	+3.510 ⇒ +3
-3.500 ⇒ -3	+3.500 ⇒ +3
-3.499... ⇒ -3	+3.499... ⇒ +3
-2.500 ⇒ -2	+2.500 ⇒ +2
-2.499... ⇒ -2	+2.499... ⇒ +2

7.7. Special Registers

GnuCOBOL, like other COBOL dialects, includes a number of data items that are automatically available to a programmer without the need to actually define them in the data division. COBOL refers to such items as registers or special registers. The special registers available to a GnuCOBOL program are as follows:

`COB-CRT-STATUS`
: PIC 9(4) — This is the default data item allocated for use by the `ACCEPT` *`data-item`* statement (see [ACCEPT data-item], page 272), if no `CRT STATUS` (see [SPECIAL-NAMES], page 92) clause was specified..

`DEBUG-ITEM`
: Group Item — A group item in which debugging information generated by a `USE FOR DEBUGGING` section in the declaratives area of the procedure division will place information documenting why the `USE FOR DEBUGGING` procedure was invoked. Consult the `DECLARATIVES` (see [DECLARATIVES], page 254) documentation for information on the structure of this register.

`LINAGE-COUNTER`
: `BINARY-LONG SIGNED` — An occurrence of this register exists for each selected file having a `LINAGE` (see [File/Sort-Description], page 124) clause. If there are multiple files whose file descriptions have `LINAGE` clauses, any explicit references to this register will require qualification (using `OF file-name`). The value of this register will be the current logical line number within the page body. The value of this register cannot be modified.

`LINE-COUNTER`
: `BINARY-LONG SIGNED` — An occurrence of this register exists for each report defined in the program (via an `RD` (see [REPORT SECTION], page 135)). If there are multiple reports, any explicit references to this register not made in the report section will require qualification (`OF report-name`). The value of this register will be the current logical line number on the current page. The value of this register cannot be modified.

`NUMBER-OF-CALL-PARAMETERS`
: `BINARY-LONG SIGNED` — This register contains the number of arguments passed to a subroutine — the same value that would be returned by the `C$NARG` built-in system subroutine (see [C$NARG], page 552). Its value will be zero when referenced in a main program. This register, when referenced from within a user-defined function, returns a value of one (‘`1`’) if the function has any number of arguments and a zero if it has no arguments.

`PAGE-COUNTER`
: `BINARY-LONG SIGNED` — An occurrence of this register exists for each report having an `RD` (see [REPORT SECTION], page 135). If there are multiple such reports, any explicit references to this register not made in the report section will require qualification (`OF report-name`). The value of this register will be the current report page number. The value of this register cannot be modified.

`RETURN-CODE`

`BINARY-LONG SIGNED` — This register provides a numeric data item into which a subroutine may `MOVE` (see [MOVE], page 352) a value (which will then be available to the calling program) prior to transferring control back to the program that called it, or into which a main program may `MOVE` a value before returning control to the operating system. Many built-in subroutines will return a value using this register. These values are — by convention — used to signify success (usually with a value of 0) or failure (usually with a non-zero value) of the process the program was attempting to perform. This register may also be modified by a subprogram as a result of that subprogram's use of the `RETURNING` (see [PROCEDURE DIVISION RETURNING], page 252) clause.

`SORT-RETURN`

`BINARY-LONG SIGNED` — This register is used to report the success/fail status of a `RELEASE` (see [RELEASE], page 371) or `RETURN` (see [RETURN], page 373) statement. A value of 0 is reported on success. A value of 16 denotes failure. An `AT END` (see [AT END + NOT AT END], page 256) condition on a `RETURN` is not considered a failure.

`WHEN-COMPILED`

`PIC X(16)` — This register contains the date and time the program was compiled in the format '`mm/dd/yyhh.mm.ss`'. Note that only a two-digit year is provided.

LENGTH OF Syntax

```
LENGTH OF numeric-literal-1 | identifier-1
~~~~~~
```

Alphanumeric literals and identifiers may optionally be prefixed with the `LENGTH OF` clause. The compile-time value generated by this clause will be the number of bytes in the alphanumeric literal or the defined size (in bytes) of the identifier.

1. The reserved word `OF` is optional and may be omitted. The presence or absence of this word has no effect upon the program. Here is an example. The following two GnuCOBOL statements both display the same result (27):

```
01  Demo-Identifier          PIC X(27).
...
    DISPLAY LENGTH OF "This is a LENGTH OF Example"
    DISPLAY LENGTH OF Demo-Identifier
```

2. The `LENGTH OF` clause on a literal or identifier reference may generally be used anywhere a numeric literal might be specified, with the following exceptions:
   - As part of the `FROM` clause of a `WRITE` (see [WRITE], page 417) or `RELEASE` statement (see [RELEASE], page 371).
   - As part of the `TIMES` clause of a `PERFORM` statement (see [PERFORM], page 360).

7.8. GnuCOBOL Statements

7.8.1. ACCEPT

7.8.1.1. ACCEPT FROM CONSOLE

ACCEPT FROM CONSOLE Syntax

```
ACCEPT { identifier-1 }   [ FROM mnemonic-name-1 ]
       { OMITTED      }

[ END-ACCEPT ]
```

This format of the `ACCEPT` statement is used to read a value from the console window or the standard input device and store it into a data item (*identifier-1*).

1. If no `FROM` clause is specified, `FROM CONSOLE` is assumed.
2. The specified *mnemonic-name-1* must either be one of the built-in device names `CONSOLE`, `STDIN`, `SYSIN` or `SYSIPT`, or a user-defined (see [SPECIAL-NAMES], page 92) mnemonic name *attached* to one of those four device names.
3. Input will be read either from the console window (`CONSOLE`) or from the system-standard input (pipe 0 = `STDIN`, `SYSIN` or `SYSIPT`) and will be saved in *identifier-1*.
4. If *identifier-1* is a numeric data item, the character value read from the console or standard-input device will be parsed according to the rules for input to the `NUMVAL` intrinsic function (see [NUMVAL], page 496), except that none of the trailing sign formats are honoured.

7.8.1.2. ACCEPT FROM COMMAND-LINE

ACCEPT FROM COMMAND-LINE Syntax

```
ACCEPT identifier-1
~~~~~~
       FROM { COMMAND-LINE                                  }
       ~~~~ { ~~~~~~~~~~~~                                  }
            { ARGUMENT-NUMBER                               }
            { ~~~~~~~~~~~~~~~                               }
            { ARGUMENT-VALUE                                }
            { ~~~~~~~~~~~~~~                                }
            [ ON EXCEPTION imperative-statement-1 ]
              ~~~~~~~~~
            [ NOT ON EXCEPTION imperative-statement-2 ]
              ~~~    ~~~~~~~~~

[ END-ACCEPT ]
  ~~~~~~~~~~
```

This format of the `ACCEPT` statement is used to retrieve information from the program's command line.

1. The reserved word `ON` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. When you accept from the `COMMAND-LINE` option, you will retrieve the entire set of arguments entered on the command line that executed the program, exactly as they were specified. Parsing that returned data into its meaningful information will be your responsibility.
3. Using `COMMAND-LINE` or `ARGUMENT-VALUE` in a *nix based platform and that includes Linux, OSX, BSD and under windows running msys or MinGW etc, the shell process will expand any arguments that have a '`*`' in the list — such as '`a*`', '`abc*.*`', etc. — and create a list of all files that match the pattern. To avoid this if not wanted, put all such argument within quotes, *e.g.*, `progundertest "a*" b c d "ef*" "*hg"` and the text within quotes will be passed verbatim to the program (as in the example `progundertest`).
4. By accepting from `ARGUMENT-NUMBER`, you will be asking the GnuCOBOL run-time system to parse the arguments from the command line and return the number of arguments found. Parsing will be conducted according to the following rules:
   A. Arguments will be separated by treating spaces and/or tab characters as the delimiters between arguments. The number of such delimiters separating two non-blank argument values is irrelevant.
   B. Strings enclosed in double-quote characters ('`"`') will be treated as a single argument, regardless of how many spaces or tab characters (if any) might be embedded within the quotation characters.

C. On Windows systems, single-quote, or apostrophe, characters (‘’) will be treated just like any other data character and will *not* delineate argument strings.

5. By accepting from `ARGUMENT-VALUE`, you will be asking the GnuCOBOL run-time system to parse the arguments from the command line and return the "current" argument. You specify which argument number is "current" via the `ARGUMENT-NUMBER` option on the `DISPLAY` statement (see [DISPLAY UPON COMMAND-LINE], page 307). Parsing of arguments will be conducted according to the rules set forth above.
6. The optional `ON EXCEPTION` and `NOT ON EXCEPTION` clauses may be used to detect and react to the failure or success, respectively, of an attempt to retrieve an `ARGUMENT-VALUE`. See [ON EXCEPTION + NOT ON EXCEPTION], page 260, for additional information.

7.8.1.3. ACCEPT FROM ENVIRONMENT

ACCEPT FROM ENVIRONMENT Syntax

```
ACCEPT identifier-1
~~~~~~
       FROM { ENVIRONMENT-VALUE            }
       ~~~~ { ~~~~~~~~~~~~~~~~            }
            { ENVIRONMENT { literal-1    } }
            { ~~~~~~~~~~~ { identifier-1 } }
     [ ON EXCEPTION imperative-statement-1 ]
        ~~~~~~~~~
     [ NOT ON EXCEPTION imperative-statement-2 ]
       ~~~    ~~~~~~~~~
[ END-ACCEPT ]
  ~~~~~~~~~~
```

This format of the `ACCEPT` statement is used to retrieve environment variable values.

1. The reserved word `ON` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. By accepting from `ENVIRONMENT-VALUE`, you will be asking the GnuCOBOL run-time system to retrieve the value of the environment variable whose name is currently in the `ENVIRONMENT-NAME` register. A value may be placed into the `ENVIRONMENT-NAME` register using the `ENVIRONMENT-NAME` option of the `DISPLAY` statement (see [DISPLAY UPON ENVIRONMENT-NAME], page 308).
3. A simpler approach to retrieving an environment variables value is to use the `ENVIRONMENT` option, where you specify the environment variable whose value is to be retrieved right on the `ACCEPT` statement itself.
4. The optional `ON EXCEPTION` and `NOT ON EXCEPTION` clauses may be used to detect and react to an attempt to retrieve the value of a non-existent environment variable or the successful retrieval of an environment variable's value, respectively. See [ON EXCEPTION + NOT ON EXCEPTION], page 260, for additional information.

7.8.1.4. ACCEPT data-item

ACCEPT data-item Syntax

```
ACCEPT { identifier-1 }
~~~~~~

       { OMITTED      }
         ~~~~~~~

                      [{ FROM EXCEPTION-STATUS }]
                         ~~~~ ~~~~~~~~~~~~~~~~

                      [{ FROM CRT ] [ MODE IS BLOCK ]}
                         ~~~~ ~~~     ~~~~    ~~~~~

       [ AT { | LINE NUMBER { integer-1    }               | } ]
         ~~ { | ~~~~        { identifier-2 }               | }
            { | COLUMN|COL|POSITION|POS NUMBER { integer-2    }
            { | ~~~~~~ ~~~ ~~~~~~~~ ~~~        { identifier-3 }
            {                                               }
            { { integer-3    }                              }
            { { identifier-4 }                              }

       [ WITH [ Attribute-Specification ]...
         ~~~~
              [ LOWER|UPPER ]
                ~~~~~ ~~~~~
              [ SCROLL { UP   } [ { integer-4    } LINE|LINES ] ]
                ~~~~~~ { ~~   }   { identifier-5 }
                       { DOWN }
                         ~~~~
              [ TIMEOUT|TIME-OUT AFTER { integer-5    } ]
                ~~~~~~~ ~~~~~~~~       { identifier-6 }
              [ CONVERSION ]
                ~~~~~~~~~~
              [ UPDATE ]
                ~~~~~~
              [ SIZE  { integer-6    } ]
                ~~~~  { identifier-7 }
       [ ON EXCEPTION imperative-statement-1 ]
            ~~~~~~~~~
       [ NOT ON EXCEPTION imperative-statement-2 ]
         ~~~    ~~~~~~~~~

[ END-ACCEPT ]
  ~~~~~~~~~~
```

The `FROM CRT`, `MODE IS BLOCK` and `CONVERSION` clauses are syntactically recognized but are otherwise non-functional.

This format of the ACCEPT statement is used to retrieve data from a working storate item or a formatted console window screen.

1. The reserved words AFTER, IS, NUMBER and ON are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved words COLUMN, COL and POSITION are interchangeable.
3. The reserved words TIMEOUT and TIME-OUT are interchangeable.
4. If *identifier-1* is defined in the SCREEN SECTION (see [SCREEN SECTION], page 143), any AT, *Attribute-Specification*, LOWER, UPPER or SCROLL clauses will be ignored. In these cases, an implied DISPLAY (see [DISPLAY data-item], page 311) of *identifier-1* will occur before input is accepted. Coding an explicit DISPLAY identifier-1 before an ACCEPT identifier-1 is redundant and will incur the performance penalty of painting the screen contents twice.
5. The various AT clauses provide a means of positioning the cursor to a specific spot on the screen before the screen is read. One or the other (but not both) may be used, as follows:
   A. The LINE and COLUMN clauses provide one mechanism for specifying the line and column position to which the cursor will be positioned before allowing the user to enter data. In the absence of one or the other, a value of 1 will be assumed for the one that is missing. The author's personal preference, however, is to explicitly code both.
   B. The *literal-3* or *identifier-4* value, if specified, must be a four- or six-digit value with the 1st half of the number indicating the line where the cursor should be positioned and the second half indicating the column. You may code only one of each clause on any ACCEPT.
6. WITH options (including the various individual *Attribute-Specifications*) should be coded only once.
7. The following *Attribute-Specification* clauses are allowed on the ACCEPT statement; these are the same as those allowed for SCREEN SECTION data items. A particular *Attribute-Specification* may be used only once in any ACCEPT:
   - AUTO (see [AUTO], page 155), AUTO-SKIP (see [AUTO-SKIP], page 156), AUTOTERMINATE (see [AUTOTERMINATE], page 157), TAB
   - BACKGROUND-COLOR (see [BACKGROUND-COLOR], page 158)
   - BEEP (see [BEEP], page 161), BELL (see [BELL], page 162)
   - BEFORE TIME (see [BEFORE TIME], page 159)
   - BLINK (see [BLINK], page 165)
   - FOREGROUND-COLOR (see [FOREGROUND-COLOR], page 173)
   - FULL (see [FULL], page 175), LENGTH-CHECK (see [LENGTH-CHECK], page 182)
   - HIGHLIGHT (see [HIGHLIGHT], page 178)
   - LEFTLINE (see [LEFTLINE], page 181)
   - LOWER (see [LOWER], page 185)

- `LOWLIGHT` (see [LOWLIGHT], page 186)
- `NO UPDATE` (see [NO UPDATE], page 189)
- `OVERLINE` (see [OVERLINE], page 193)
- `PROMPT` (see [PROMPT], page 202)
- `PROTECTED` (see [PROTECTED], page 203)
- `REQUIRED` (see [REQUIRED], page 206), `EMPTY-CHECK` (see [EMPTY-CHECK], page 169)
- `REVERSE-VIDEO` (see [REVERSE-VIDEO], page 207)
- `SCROLL DOWN` (see [SCROLL DOWN], page 210)
- `SCROLL UP` (see [SCROLL UP], page 211)
- `SIZE` (see [SIZE], page 214)
- `SECURE` (see [SECURE], page 212), `NO-ECHO` (see [NO-ECHO], page 188)
- `TIME OUT` (see [TIME OUT], page 220)
- `UPDATE` (see [UPDATE], page 230)
- `UPPER` (see [UPPER], page 231)
- `UNDERLINE` (see [UNDERLINE], page 229)

8. The `SCROLL` option will cause the entire contents of the screen to be scrolled `UP` or `DOWN` by the specified number of lines before any value is displayed on the screen. It is syntactically allowable to specify a `SCROLL UP` clause as well as a `SCROLL DOWN` clause. In such an instance, it is the last one specified that will be honoured. If no `LINES` specification is made, `1 LINE` will be assumed.
9. The `TIMEOUT` option will cause the `ACCEPT` to wait no more than the specified number of seconds for input. The wait count may be specified as a positive integer or a numeric data item with a positive value.
10. The `UPDATE` option will enable the supplied data field to be updated having been displayed on screen prior to data being entered by overwriting, if needed. When this option is *not* used the input field is cleared prior to input and this is the default but can be changed by the use of compiler steering command -faccept-with-update that can be input when starting the compiler or included in the configuration file e.g., default.conf used when selected by default `-std=default`. For more information see ⟨undefined⟩ [cobc - The GnuCOBOL Compiler], page ⟨undefined⟩, option switches) and [Compiler Configuration Files], page 649.
11. This format of the `ACCEPT` statement will be terminated by any of the following events:
    A. When the `Enter` key is pressed.
    B. Expiration of the `TIMEOUT` timer — this will be treated as if the `Enter` key had been pressed with no data being entered.
    C. When a function key (`Fn`) is pressed.
    D. The pressing of the `PgUp` or `PgDn` keys, if the `COB_SCREEN_EXCEPTIONS` run-time environment variable (see [Run Time Environment Variables], page 658) is set to any non-blank value.

E. The pressing of the `Esc` key if *both* the `COB_SCREEN_ESC` run-time environment variable as well as `COB_SCREEN_EXCEPTIONS` run-time environment variable are set to any non-blank value.

F. The pressing of the `Up-arrow`, `Down-Arrow` or `PrtSc` (Print Screen) keys. These keys are not detectable on Windows systems, however.

12. The following apply when *identifier-1* is defined in the `SCREEN SECTION`:

    A. Alphanumeric data entered into *identifier-1* or any screen data item subordinate to it *must* be consistent with the `PICTURE` (see [PICTURE], page 194) clause of that item. This will be enforced at runtime by the `ACCEPT` statement.

    B. If *identifier-1* or any screen data item subordinate to it are defined as numeric, entered data must be acceptable as `NUMVAL` intrinsic function (see [NUMVAL], page 496) input (no decimal points are allowed, however). The value stored into the screen data item will be as if the input were passed to that function.

    C. If *identifier-1* or any screen data item subordinate to it are defined as numeric edited, entered data must be acceptable as `NUMVAL-C` intrinsic function (see [NUMVAL-C], page 497) input (again, no decimal points are allowed). The value stored into the screen data item will be as if the input were passed to that function.

13. The following apply when *identifier-1* is *not* defined in the `SCREEN SECTION`:

    A. Alphanumeric data entered into *identifier-1 should* be consistent with the `PICTURE` (see [PICTURE], page 194) clause of that item, although that will not be enforced by the `ACCEPT` statement. You may use `Class Conditions` (see [Class Conditions], page 46) after the data is accepted to enforce the data type.

    B. If *identifier-1* is defined as numeric, entered data must be acceptable as `NUMVAL` intrinsic function (see [NUMVAL], page 496) input (no decimal points are allowed, however). The value stored into *identifier-1* will be as if the input were passed to that function.

    C. If *identifier-1* is defined as numeric edited, entered data must be acceptable as `NUMVAL-C` intrinsic function (see [NUMVAL-C], page 497) input (again, no decimal points are allowed). The value stored into *identifier-1* will be as if the input were passed to that function.

14. The optional `ON EXCEPTION` and `NOT ON EXCEPTION` clauses may be used to detect and react to the failure or success, respectively, of the screen I/O attempt. See [ON EXCEPTION + NOT ON EXCEPTION], page 260, for additional information.

    After this format of the `ACCEPT` statement is executed, the program's `CRT STATUS` (see [SPECIAL-NAMES], page 92) identifier will be populated with one of the following:

Code	**Meaning**
0000	`ENTER` key pressed
1001–1064	`F1`–`F64`, respectively, were pressed
2001	`PgUp` was pressed
2002	`PgDn` was pressed
2003	`Up-Arrow` was pressed

2004	`Down-Arrow` was pressed
2005	`Esc` was pressed
2006	`PrtSc` (Print Screen) was pressed
2007	`Tab`
2008	`Back Tab`
2009	`Key Left`
2010	`Key Right`
2011	`Insert` Key on accept omitted
2012	`Delete` Key on accept omitted
2013	`Backspace` Key on accept omitted
2014	`Home` Key on accept omitted
2015	`End` Key on accept omitted
2040-	2095 Exception keys for Mouse Handling
2040	Mouse Move
2041	Left Pressed
2042	Left Released
2043	Left Dbl Click
2044	Mid Pressed
2045	Mid Released
2046	Mid Dbl Click
2047	Right Pressed
2048	Right Released
2049	Right Dbl Click
2050	Shift Move
2051	Shift Left Pressed
2052	Shift Left Released
2053	Shift Left Dbl Click
2054	Shift Mid Pressed
2055	Shift Mid Released
2056-	Shift Mid Dbl Click
2057	Shift Right Pressed
2058	Shift Right Released
2059	Shift Right Dbl Click
2060	Ctrl Move
2061	Ctrl Left Pressed
2062	Ctrl Left Released
2063	Ctrl Left Dbl Click
2064	Ctrl Mid Pressed
2065	Ctrl Mid Released
2066	Ctrl Mid Dbl Click
2067	Ctrl Right Pressed
2068	Ctrl Right Released
2069	Ctrl Right Dbl Click
2070	Alt Move
2071	Alt Left Pressed
2072	Alt Left Released

2073-	Alt Left Dbl Click
2074	Alt Mid Pressed
2075	Alt Mid Released
2076	Alt Mid Dbl Click
2077	Alt Right Pressed
2078	Alt Right Released
2079	Alt Right Dbl Click
2080	Wheel Up
2081	Wheel Down
2082	Wheel Left
2083	Wheel Right
2084	Shift Wheel Up
2085	Shift Wheel Down
2086	Shift Wheel Left
2087	Shift Wheel Right
2088	Ctrl Wheel Up
2089	Ctrl Wheel Down
2090	Ctrl Wheel Left
2091	Ctrl Wheel Right
2092	Alt Wheel Up
2093	Alt Wheel Down
2094	Alt Wheel Left
2095	Alt Wheel Right
	Input validation
8000	NO Field
8001	Time Out
	Other errors
9000	Fatal
9001	Max Field

15. The actual key pressed to generate a function key (`Fn`) will depend on the type of terminal device you're using (PC, Macintosh, VT100, etc.) and what type of enhanced display driver was configured with the version of GnuCOBOL you're using.

    For example, on a GnuCOBOL build for a Windows PC using MinGW and "PD-Curses", `F1`–`F12` are the actual F-keys on the PC keyboard, `F13`–`F24` are entered by shifting the F-keys, `F25`–`F36` are entered by holding Ctrl while pressing an F-key and `F37`–`F48` are entered by holding Alt while pressing an F-key. On the other hand, a GnuCOBOL implementation built for Windows using Cygwin and NCurses treats the PCs `F1`–`F12` keys as the actual `F1`–`F12`, while shifted F-keys will enter `F11`–`F20`. With Cygwin/NCurses, Ctrl- and Alt-modified F-keys aren't recognized, nor are `Shift-F11` or `Shift-F12`.

    Mouse Key codes are populated only if mouse management has been enabled. To enable mouse it is first necessary to set `COB_MOUSE_FLAGS` (either externally via terminal command, or internally via `SET ENVIRONMENT` to the applicable ?mouse mask? (specifying which activities you wish the program to detect). Here is an example of setting the mask from a COBOL program:

```
COPY screenio.cpy.

01  mouse-flags PIC 9(4).

...

    COMPUTE mouse-flags = COB-AUTO-MOUSE-HANDLING
                        + COB-ALLOW-LEFT-DOWN
                        + COB-ALLOW-MIDDLE-DOWN
                        + COB-ALLOW-RIGHT-DOWN

    SET ENVIRONMENT "COB_MOUSE_FLAGS" TO mouse-flags.
```

16. Once that has been done, every (extended) ACCEPT, will return a value in COB_CRT_STATUS reflecting mouse activity, when such activity occurs. The applicable values are shown in screenio.cpy under ?Exception keys for mouse handling?. If you define a variable in SPECIAL NAMES as follows:

    ```
    SPECIAL-NAMES.
       CURSOR IS data-name.   *> where data-name is PIC 9(4) or 9(6).
    ```

17. The cursor or mouse position will be returned as well. The position is expressed as row and column (rrcc or rrrccc).
18. Numeric keypad keys are not recognizable on Windows MinGW/PDCurses builds of GnuCOBOL, regardless of the number lock settings. Windows Cygwin/NCurses builds recognize numeric keypad inputs properly. Although not tested during the preparation of this documentation, I would expect native Windows builds using PDCurses to behave as MinGW builds do and native Unix builds using NCurses to behave as do Cygwin builds.
19. The optional `EXCEPTION-STATUS` clause may be used to detect exceptions from a prior arithmetic verb such as COMPUTE to recover any errors produced. These are recovered using the function `EXCEPTION-STATUS`.

7.8.1.5. ACCEPT FROM DATE/TIME

ACCEPT FROM DATE/TIME Syntax

```
ACCEPT identifier-1 FROM { DATE [ YYYYMMDD ] }
~~~~~~                ~~~~ { ~~~~   ~~~~~~~~   }
                           { DAY [ YYYYDDD ]   }
                           { ~~~   ~~~~~~~     }
                           { DAY-OF-WEEK       }
                           { ~~~~~~~~~~~       }
                           { TIME              }
                           { ~~~~              }
                           { MICROSECOND-TIME  }
                           { ~~~~~~~~~~~~~~~~  }
[ END-ACCEPT ]
  ~~~~~~~~~~
```

This format of the `ACCEPT` statement is used to retrieve the current system date, time or current day of the week and store it into a data item.

1. The data retrieved from the system and the format in which it is structured will vary, as follows:

Syntax	Data Retrieved	Format
`DATE`	Current date in Gregorian form	yymmdd
`DATE YYYYMMDD`	Current date in Gregorian form	yyyymmdd
`DAY`	Current date in Julian form	yyddd
`DAY YYYYDDD`	Current date in Julian form	yyyyddd
`DAY-OF-WEEK`	Current day within a week where 1 = Monday.., 7 = Sunday	d
`TIME`	Time, including hundredths of a second (n)	hhmmssnn
`MICROSECOND-TIME`	Time, including micro seconds (u)	hhmmssuuuuuu

2. When using one of `--std=acu` or `--std=acu-strict`, ACCEPT FROM TIME data-item with data-item providing more than 12 digits behaves as if ACCEPT FROM MICROSECOND-TIME data-item is used (six decimal places for fractional seconds).
3. Consider using the standard `FUNCTION FORMATTED-CURRENT-DATE` if you need a high precision (up to eight decimal places for fractional seconds).

7.8.1.6. ACCEPT FROM Screen-Info

ACCEPT FROM Screen-Info Syntax

```
ACCEPT identifier-1
~~~~~~

       FROM { LINES|LINE-NUMBER }
       ~~~~ { ~~~~~ ~~~~~~~~~~~ }
            { COLS|COLUMNS      }
            { ~~~~ ~~~~~~~      }
            { ESCAPE KEY        }
              ~~~~~~ ~~~

[ END-ACCEPT ]
  ~~~~~~~~~~
```

This format of the `ACCEPT` statement is used to retrieve information about the console window or about the user's interactions with it.

1. The reserved words `LINES` and `LINE-NUMBER` are interchangeable.
2. The reserved words `COLS` and `COLUMNS` are interchangeable.
3. The following points pertain to the use of the `LINES` and `COLUMNS` options:
   A. The `LINES` and `COLUMNS` options will retrieve the respective components of the size of the console display.
   B. When the console is running in a windowed environment, this will be the sizing of the window in which the program is executing, in terms of horizontal (`COLUMNS`) or vertical (`LINES`) character counts — not pixels.
   C. When the system is not running a windowing environment, the physical console screen attributes will be returned.
   D. Values of 0 will be returned if GnuCOBOL was not generated to include screen I/O.
   E. See the documentation on the `CBL_GET_SCR_SIZE` built-in system subroutine (see [CBL_GET_SCR_SIZE], page 585) for another way to retrieve this information.
4. The `ESCAPE KEY` option may be used after the `ACCEPT FROM Screen-Info` statement (see [ACCEPT FROM Screen-Info], page 280) has executed. The result returned will be the four-digit `CRT STATUS` (see [SPECIAL-NAMES], page 92) identifier value. See [CRT STATUS Codes], page 275, for the specific code values.

7.8.1.7. ACCEPT FROM Runtime-Info

ACCEPT FROM Runtime-Info Syntax

```
ACCEPT identifier-1
~~~~~~

       FROM { EXCEPTION STATUS }
       ~~~~ { ~~~~~~~~~ ~~~~~~ }
            { USER NAME        }
              ~~~~ ~~~~

[ END-ACCEPT ]
  ~~~~~~~~~~
```

This format of the `ACCEPT` statement is used to retrieve run-time information such as the most-recent error exception code and the current user's user name.

1. The following points pertain to the use of the `EXCEPTION STATUS` option:
   A. *identifier-1* must be defined as a `PIC X(4)` item.
   B. See [Error Exception Codes], page 451, for a complete list of the exception codes and their meanings.
   C. An alternative to the use of `ACCEPT FROM Runtime-Info` is to use the `EXCEPTION-STATUS` intrinsic function (see [EXCEPTION-STATUS], page 451).
2. The following points pertain to the use of the `USER NAME` option:
   A. The returned result is the userid that was used to login to the system with, and not any actual first and/or last name of the user in question (unless, of course, that is the information used as a logon id). It is not the PID or UID numbers but the name associated with the UID under *nix based systems.
   B. *identifier-1* should be defined large enough to receive the longest user-name on the system.
   C. If insufficient space is allocated, the returned value will be truncated.
   D. If excess space is allocated, the returned value will be padded with spaces (to the right).

7.8.1.8. ACCEPT OMITTED

ACCEPT OMITTED Syntax

```
ACCEPT OMITTED
~~~~~~

1.    For console :  See 6.17.1.1 (ACCEPT FROM CONSOLE Syntax)

2.    For Screen  :  See 6.17.1.4 (ACCEPT screen-data-item Syntax)

[ END-ACCEPT ]
  ~~~~~~~~~~
```

This format of the `ACCEPT` statement will wait for a keyboard event that terminates input; function keys, or Enter/Return, among others. CRT STATUS (COB-CRT-STATUS `CRT STATUS` (see [SPECIAL-NAMES], page 92) if not explicitly defined) is set with the keycode, listed in copy/screenio.cpy. It also handles a few other keycode terminations not normally used to complete an extended accept.

1. The following are examples of keycodes that can be used:

   ```
   COB-SCR-INSERT
   COB-SCR-DELETE
   COB-SCR-BACKSPACE
   COB-SCR-KEY-HOME
   COB-SCR-KEY-END
   ```

2. You can used extended attributes, useful for setting timeouts or positioning.

7.8.1.9. ACCEPT FROM EXCEPTION STATUS

ACCEPT FROM EXCEPTION STATUS Syntax

```
ACCEPT exception-status-pic-9-4   FROM EXCEPTION STATUS
~~~~~~                            ~~~~ ~~~~~~~~~ ~~~~~~

[ END-ACCEPT ]
  ~~~~~~~~~~
```

This format of the `ACCEPT` statement will receive the status for any exceptions resulting from a previous valid verb.

1. The following is an example of usage:

```
In WS:
01  exception-status  pic 9(4).
..
In PD:

ACCEPT unexpected-rounding  FROM EXCEPTION STATUS
IF unexpected-rounding NOT EQUAL "0000" THEN
   DISPLAY "Unexpected rounding. Code " unexpected-rounding
            UPON SYSERR
END-IF
```

7.8.2. ADD

7.8.2.1. ADD TO

ADD TO Syntax

```
ADD { literal-1    }...
~~~ { identifier-1 }

    TO { identifier-2
    ~~
       [ ROUNDED [ MODE IS { AWAY-FROM-ZERO         } ] ] }...
         ~~~~~~~   ~~~~    { ~~~~~~~~~~~~~~         }
                           { NEAREST-AWAY-FROM-ZERO }
                           { ~~~~~~~~~~~~~~~~~~~~~~ }
                           { NEAREST-EVEN           }
                           { ~~~~~~~~~~~~           }
                           { NEAREST-TOWARD-ZERO    }
                           { ~~~~~~~~~~~~~~~~~~     }
                           { PROHIBITED             }
                           { ~~~~~~~~~~             }
                           { TOWARD-GREATER         }
                           { ~~~~~~~~~~~~~~         }
                           { TOWARD-LESSER          }
                           { ~~~~~~~~~~~~~          }
                           { TRUNCATION             }
                             ~~~~~~~~~~

   [ ON SIZE ERROR imperative-statement-1 ]
       ~~~~ ~~~~~

   [ NOT ON SIZE ERROR imperative-statement-2 ]
     ~~~    ~~~~ ~~~~~

[ END-ADD ]
  ~~~~~~~
```

This format of the `ADD` statement generates an intermediate arithmetic sum of the values of all *identifier-1* and *literal-1*) items. The value of each *identifier-2* will be replaced, in turn, by the sum of that *identifier-2*s value and the intermediate sum.

1. The reserved words `IS` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. Both *identifier-1* and *identifier-2* must be numeric unedited data items while *literal-1* must be a numeric literal.
3. An *identifier-1* data item may also be coded as an *identifier-2*. Note, however, that the value of such a data item will therefore be included *twice* in the result.

4. The contents of each *identifier-1* will remain unchanged by this statement.
5. The optional `ROUNDED` (see [ROUNDED], page 261) clause available to each *identifier-2* will control how non-integer results will be saved.
6. The optional `ON SIZE ERROR` and `NOT ON SIZE ERROR` clauses may be used to detect and react to the failure or success, respectively, of an attempt to perform a calculation. In this case, failure is defined as being an *identifier-2* with an insufficient number of digit positions available to the left of any implied decimal point. See [ON SIZE ERROR + NOT ON SIZE ERROR], page 261, for additional information.

7.8.2.2. ADD GIVING

ADD GIVING Syntax

```
ADD { literal-1    }...
~~~ { identifier-1 }

  [ TO identifier-2 ]
    ~~
    GIVING { identifier-3
    ~~~~~~
      [ ROUNDED [ MODE IS { AWAY-FROM-ZERO         } ] ] }...
        ~~~~~~~   ~~~~    { ~~~~~~~~~~~~~~         }
                          { NEAREST-AWAY-FROM-ZERO }
                          { ~~~~~~~~~~~~~~~~~~~~~~ }
                          { NEAREST-EVEN           }
                          { ~~~~~~~~~~~~           }
                          { NEAREST-TOWARD-ZERO    }
                          { ~~~~~~~~~~~~~~~~~~     }
                          { PROHIBITED             }
                          { ~~~~~~~~~~             }
                          { TOWARD-GREATER         }
                          { ~~~~~~~~~~~~~~         }
                          { TOWARD-LESSER          }
                          { ~~~~~~~~~~~~~          }
                          { TRUNCATION             }
                            ~~~~~~~~~~

  [ ON SIZE ERROR imperative-statement-1 ]
      ~~~~ ~~~~~

  [ NOT ON SIZE ERROR imperative-statement-2 ]
    ~~~    ~~~~ ~~~~~

[ END-ADD ]
  ~~~~~~~
```

This format of the `ADD` statement generates the arithmetic sum of the values of all *identifier-1*, *literal-1*) and *identifier-2* (if any) items and then saves that sum to each *identifier-3*.

1. The reserved words `IS` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. Both *identifier-1* and *identifier-2* must be numeric unedited data items while *literal-1* must be a numeric literal; *identifier-3* may be either a numeric or numeric edited data item.
3. An *identifier-1* or *identifier-2* data item may be used as an *identifier-3*, if desired.
4. The contents of each *identifier-1* and *identifier-2* will remain unchanged by this state-

ment, unless they happen to also be specified as an *identifier-3*.

5. The current value in each *identifier-3* at the start of the statement's execution is irrelevant, since the contents of each *identifier-3* will simply be replaced with the computed sum.
6. The optional `ROUNDED` (see [ROUNDED], page 261) clause available to each *identifier-3* will control how non-integer results will be saved.
7. The optional `ON SIZE ERROR` and `NOT ON SIZE ERROR` clauses may be used to detect and react to the failure or success, respectively, of an attempt to perform a calculation. In this case, failure is defined as being an *identifier-3* with an insufficient number of digit positions available to the left of any implied decimal point. See [ON SIZE ERROR + NOT ON SIZE ERROR], page 261, for additional information.

7.8.2.3. ADD CORRESPONDING

ADD CORRESPONDING Syntax

```
ADD CORRESPONDING identifier-1
~~~

    TO identifier-2
    ~~

  [ ROUNDED [ MODE IS { AWAY-FROM-ZERO         } ] ]
    ~~~~~~~   ~~~~    { ~~~~~~~~~~~~~~         }
                      { NEAREST-AWAY-FROM-ZERO }
                      { ~~~~~~~~~~~~~~~~~~~~~~ }
                      { NEAREST-EVEN           }
                      { ~~~~~~~~~~~~           }
                      { NEAREST-TOWARD-ZERO    }
                      { ~~~~~~~~~~~~~~~~~~     }
                      { PROHIBITED             }
                      { ~~~~~~~~~~             }
                      { TOWARD-GREATER         }
                      { ~~~~~~~~~~~~~~         }
                      { TOWARD-LESSER          }
                      { ~~~~~~~~~~~~~          }
                      { TRUNCATION             }
                        ~~~~~~~~~~

  [ ON SIZE ERROR imperative-statement-1 ]
      ~~~~ ~~~~~

  [ NOT ON SIZE ERROR imperative-statement-2 ]
    ~~~    ~~~~ ~~~~~

[ END-ADD ]
  ~~~~~~~
```

This format of the `ADD` statement generates code equivalent to individual `ADD TO` (see [ADD TO], page 284) statements for corresponding matches of data items found subordinate to the two identifiers.

1. The reserved words `IS` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. Both *identifier-1* and *identifier-2* must be group items.
3. See [CORRESPONDING], page 258, for information on how corresponding matches will be found between *identifier-1* and *identifier-2*.
4. The optional `ROUNDED` (see [ROUNDED], page 261) clause available to each *identifier-3* will control how non-integer results will be saved.
5. The optional `ON SIZE ERROR` and `NOT ON SIZE ERROR` clauses may be used to detect and react to the failure or success, respectively, of an attempt to perform a calculation. In

this case, failure is defined as being an *identifier-3* with an insufficient number of digit positions available to the left of any implied decimal point. See [ON SIZE ERROR + NOT ON SIZE ERROR], page 261, for additional information.

7.8.3. ALLOCATE

ALLOCATE Syntax

```
FORMAT 1.  ALLOCATE a "BASED" ITEM.

 ALLOCATE  identifier-1
 ~~~~~~~~
           [{ INITIALIZED } ]  [ RETURNING identifier-3 ]
           [{ ~~~~~~~~~~~ } ]    ~~~~~~~~~
           [{ INITIALISED } ]
           [{ ~~~~~~~~~~~ } ]


FORMAT 2.  ALLOCATE a memory block.

 ALLOCATE   arithmetic-expression-1  CHARACTERS
 ~~~~~~~~                            ~~~~~~~~~~
           [{ INITIALIZED } [ TO { identifier-2}] ]  RETURNING identifier-3
           [{ ~~~~~~~~~~~ } [ ~~ {             }] ]  ~~~~~~~~~
           [{ INITIALISED } [    { literal-1   }] ]
           [{ ~~~~~~~~~~~ }                      ]
```

The `ALLOCATE` statement is used to dynamically allocate memory at run-time.

1. The reserved words `INITIALIZED` and `INITIALISED` are interchangeable.
2. If data-name-1 is specified, the RETURNING phrase may be omitted; otherwise, the RETURNING phrase shall be specified.
3. If used, *expression-1* must be an arithmetic expression with a non-zero positive integer value and the RETURNING phrase must be specified.
4. If used, *identifier-1* should be an 01-level item defined in working-storage or local-storage with the `BASED` (see [BASED], page 160) attribute. It may be an 01 item defined in the linkage section without the `BASED` attribute, but using such a data item is not recommended.
5. If used, *identifier-3* should be a `POINTER` (see [USAGE], page 232) data item.
6. The optional `RETURNING` clause will return the address of the allocated memory block into the specified `USAGE POINTER` *identifier-3* data item. When this option is used, knowledge of the originally-requested size of the allocated memory block will be retained by the program in case a `FREE` (see [FREE], page 331) statement is ever issued against *identifier-3*.
7. When the *identifier-1* option is used in conjunction with `INITIALIZED` (or its internationalized alternative `INITIALISED`), the allocated memory block will be initialized

as if an `INITIALIZE` *`identifier-1`* `WITH FILLER ALL TO VALUE THEN TO DEFAULT` (see [INITIALIZE], page 339) were executed.

8. When the *`expression-1`* `CHARACTERS` option is used, `INITIALIZED` will initialize the allocated memory block to binary zeros. If `INITIALIZED` is not used, the initial contents of allocated memory will be left to whatever rules of memory allocation are in effect for the operating system the program is running under.
9. There are two basic ways in which this statement is used. The simplest is:

   ```
   ALLOCATE My-01-Item
   ```

   With this form, a block of storage equal in size to the defined size of My-01-Item (which must have been defined with the `BASED` attribute) will be allocated. The address of that block of storage will become the base address of My-01-Item so that it and its subordinate data items become usable within the program.

   A second (and equivalent) approach is:

   ```
   ALLOCATE LENGTH OF My-01-Item CHARACTERS RETURNING The-Pointer
   SET ADDRESS OF My-01-Item TO The-Pointer
   ```

10. With this form My-01-Item can either be defined with the `BASED` attribute or be defined in LINKAGE SECTION. Instead of LENGTH OF My-01-Item you may also use a size smaller to the maximum field size as long as you ensure that the complete field is never used.
11. Referencing a `BASED` data item either before its storage has been allocated or after its storage has been released (via the `FREE` statement) will lead to "unpredictable results". That's how reference manuals and standards specifications talk about this situation. In the author's experience, the results are all too predictable: the program aborts from an attempt to reference an unallocated area of memory.

7.8.4. ALTER

ALTER Syntax

```
ALTER procedure-name-1 TO PROCEED TO procedure-name-2
~~~~~                  ~~
```

The `ALTER` statement was used in the early years of the COBOL language to edit the object code of a program **at execution time**, changing a `GO TO` (see [Simple GO TO], page 335) statement to branch to a spot in the program different than where the `GO TO` statement was originally compiled for.

1. The reserved words `PROCEED` and `TO` (the one *after* `PROCEED`) are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. *procedure-name-1* must contain only a single statement, and that statement must be a simple `GO TO`.
3. The effect of this statement will be as if the generated machine-language code for the `GO TO` statement in *procedure-name-1* is changed so that the `GO TO` statement now transfers control to *procedure-name-2*, rather than to whatever procedure name was specified in the program source code.
4. Support for the `ALTER` verb has been added to GnuCOBOL for the purpose of enabling GnuCOBOL to pass those National Institute of Standards and Technology (NIST) tests for the COBOL programming language that require support for `ALTER`.
5. Because of the catastrophic effect this statement has on program readability and therefore the programmer's ability to debug problems with program logic, the use of `ALTER` in new programs is **STRONGLY** discouraged.

7.8.5. CALL

CALL Syntax

```
CALL [ { STDCALL         } ] { literal-1    }
~~~~   { ~~~~~~~         }   { identifier-1 }
       { STATIC          }
       { ~~~~~~          }
       { mnemonic-name-1 }

     [ USING CALL-Argument...  ]
       ~~~~~

     [ RETURNING|GIVING identifier-2 ]
       ~~~~~~~~~ ~~~~~~

     [ ON OVERFLOW|EXCEPTION imperative-statement-1 ]
          ~~~~~~~~ ~~~~~~~~~

     [ NOT ON OVERFLOW|EXCEPTION imperative-statement-2 ]
       ~~~    ~~~~~~~~ ~~~~~~~~~

[ END-CALL ]
  ~~~~~~~~
```

CALL Argument Syntax

```
[ BY { REFERENCE } ]
     { ~~~~~~~~~ }
     { CONTENT   }
     { ~~~~~~~   }
     { VALUE     }
       ~~~~~

     { OMITTED                                                    }
     { ~~~~~~~                                                    }
     { [ UNSIGNED ] [ SIZE IS { AUTO      } ] [ { literal-2    } ] }
         ~~~~~~~~     ~~~~    { ~~~~      }     { identifier-2 }
                              { DEFAULT   }
                              { ~~~~~~~   }
                              { integer-1 }
```

The `CALL` statement is used to transfer control to a subroutine. See [Sub-Programming], page 679, for the specifics of using subprograms with GnuCOBOL programs.

1. The reserved words `BY`, `IS` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.

2. The reserved words `EXCEPTION` and `OVERFLOW` are interchangeable.
3. The reserved words `GIVING` and `RETURNING` are interchangeable.
4. The expectation is that the subroutine will eventually return control back to the calling program, at which point the `CALL`ing program will resume execution starting with the statement immediately following the `CALL`. Subprograms are not required to return to their callers, however, and are free to halt program execution if they wish.
5. The *mnemonic-name-1* / `STATIC` / `STDCALL` option, if used, affects the linkage conventions that will be used to the subroutine being called, as follows:

   `STATIC` causes the linkage to the subroutine to be performed in such a way as to require the subroutine to be statically-linked with the calling program. Note that this enables static-linking to be used on a subroutine-by-subroutine selective basis.

   `STDCALL` allows system standard calling conventions (as opposed to GnuCOBOL calling conventions) to be used when calling a subroutine. The definition of what constitutes "system standard" may vary from operating system to operating system. Use of this requires special knowledge about the linkage requirements of subroutines you are intending to `CALL`. Subroutines written in GnuCOBOL do not need this option.

   mnemonic-name-1
   allows a custom defined calling convention to be used. Such mnemonic names are defined using the `CALL-CONVENTION` (see [SPECIAL-NAMES], page 92) clause. That clause associates a decimal integer value with *mnemonic-name-1* such that the individual bits set on or off in the binary equivalent of the integer affect linkage to the subroutine as described in the following chart. Those rows of the chart marked with a "No" in the **Supported** column represent bit positions (switch settings) in the integer value that are currently accepted (to provide compatibility to other COBOL implementations) if coded, but are otherwise unsupported.

   Note that bit 0 is the right-most bit in the binary value.

Bit	Supported	Meaning if 0	Meaning if 1
0	No	Arguments will be passed in right-to-left sequence	Arguments will be passed in left-to-right sequence.
1	No	The calling program will flush processed arguments from the argument stack.	The called program (subroutine) will flush processed arguments from the argument stack.
2	Yes	The `RETURN-CODE` special register (see [Special Registers], page 265) will be updated in addition to any `RETURNING` or `GIVING` data item.	The `RETURN-CODE` special register will not be updated (but any `RETURNING` or `GIVING` data item still will).

3	Yes	If `CALL literal` is used, the subroutine will be located and linked in with the calling program at compile time or may be dynamically located and loaded at execution time, depending on compiler switch settings and operating system capabilities.	If `CALL literal` is used, the subroutine can only be located and linked with the calling program at compilation time.
4	No	OS/2 "OPTLINK" conventions will not be used to CALL the subprogram.	OS/2 "OPTLINK" conventions will be used to CALL the subprogram.
5	No	Windows 16-bit "thunking" will not be in effect.	Windows 16-bit "thunking" will be used to call the subroutine as a DLL.
6	Yes	The `STDCALL` convention will not be used.	The `STDCALL` convention, required to use the Microsoft Win32 API, will be used.

Using the `STATIC` option on a `CALL` statement is equivalent to using `CALL-CONVENTION 8` (only bit 3 set).

Using the `STDCALL` option on a `CALL` statement is equivalent to using `CALL CONVENTION 64` (only bit 6 set).

6. The value of *literal-1* or *identifier-1* is the entry-point of the subprogram you wish to call.

7. When you call a subroutine using *identifier-1*, you are forcing the runtime system to call a dynamically-loadable subprogram. The contents of *identifier-1* will be the entry-point name within that module. If this is the *first* call to any entry-point within the module being made at run-time, the contents of *identifier-1* must be the primary entry-point name of the module (which must also match the filename, minus any OS-mandated extension) of the executable file comprising the module).

8. You can force the GnuCOBOL runtime system to pre-load all dynamically-loaded modules that could ever be called by the program, at the time the program starts executing. This is accomplished through the use of the `COB_PRE_LOAD` run-time environment variable (see [Run Time Environment Variables], page 658). If used, this will only pre-load those modules invoked via `CALL literal-1`, as the runtime contents of *identifier-1* cannot be predicted.

9. If the subprogram being called is a GnuCOBOL program, and if that program had the `INITIAL` (see [IDENTIFICATION DIVISION], page 85) attribute specified on its `PROGRAM-ID` clause, all of the subprogram's data division data will be restored to its initial state each time the subprogram is executed, regardless of which entry-point within the subprogram is being referenced.

This [re]-initialization behaviour will *always* apply to any subprogram's local-storage (if any), regardless of the use (or not) of `INITIAL`.

10. The `USING` clause defines a list of arguments that may be passed from the calling program to the subprogram. The manner in which any given argument is passed to the subroutine depends upon the `BY` clause (if any) coded (or implied) for that argument, as follows:

    `BY REFERENCE`
    passes the *address* of the argument to the subprogram. If the subprogram changes the contents of that argument, the change will be "visible" to the calling program.

    `BY CONTENT`
    passes the *address* of a *copy* of the argument to the subprogram. If the subprogram changes the value of such an argument, the change only affects the copy back in the calling program, not the original version.

    `BY VALUE` passes the *actual numeric value* of the literal or identifiers contents as the argument. This feature exists to provide compatibility with C, C++ and other languages and would not normally be used when calling GnuCOBOL subprograms. Only numeric literals or numeric data items should be passed in this manner.

    If an argument lacks a `BY` clause, the most-recently encountered `BY` specification on that `CALL` statement will be assumed. If the first argument specified on a `CALL` lacks a `BY` clause, `BY REFERENCE` will be assumed.

11. No more than 251 arguments may be passed to a subroutine, unless the GnuCOBOL compiler was built with a specifically different argument limit specified for it. If you have access to the GnuCOBOL source code, you may adjust this limit by changing the value of the `COB_MAX_FIELD_PARAMS` in the `call.c` file (found in the `libcob` folder) as well as the last shown `#if MAX_CALL_FIELD_PARAMS` statement before you run `make` to build the compiler and run-time library.

12. The `RETURNING` clause allows you to specify a numeric data item into which the subroutine should return a numeric value. If you use this clause on the `CALL`, the subroutine should include a `RETURNING` (see [PROCEDURE DIVISION RETURNING], page 252) clause on its procedure division header. Of course, a subroutine may pass a value of any kind back in any argument passed `BY REFERENCE`.

13. The optional `ON OVERFLOW` and `NOT ON OVERFLOW` clauses (or `ON EXCEPTION` and `NOT ON EXCEPTION` — they are interchangeable) may be used to detect and react to the failure or success, respectively, of an attempt to `CALL` the subroutine. Failure, in this context, is defined as the inability to either locate or load the object code of the subroutine at execution time. See [ON OVERFLOW + NOT ON OVERFLOW], page 260, for additional information.

14. Call also supports using an entry point stored in a `PROGRAM-POINTER`, avoiding the dynamic runtime lookup. GnuCOBOL keeps a cache of lookups during a program run. Repeated use of a named function does not suffer much penalty, but `PROGRAM-POINTER` will be just that little bit faster. To set a `PROGRAM-POINTER` use

`SET` *`program-reference`* `TO ENTRY "`*`name`*`"` (or get the address from an API, and take part in callback programming).

15. An extension of `CALL` allows a call to a *Program-Pointer-1* which is preset using `SET` *`program-pointer-1`* `TO ENTRY` *`x`*. Additional the `RETURNING` clause may return a data pointer or a `PROGRAM-POINTER`

7.8.6. CANCEL

CANCEL Syntax

```
CANCEL { literal-1    }...
~~~~~~ { identifier-1 }
```

The `CANCEL` statement unloads the dynamically-loadable subprogram module containing the entry-point specified as *literal-1* or *identifier-1* from memory.

1. If a dynamically-loadable module unloaded by the `CANCEL` statement is subsequently re-executed, all data division storage for that module will once again be in its initial state.
2. Whether the `CANCEL` statement actually physically unloads a dynamically-loaded module or simply marks it as logically-unloaded depends on the use and value of the `COB_PHYSICAL_CANCEL` run-time environment variable (see [Run Time Environment Variables], page 658).

7.8.7. CLOSE

CLOSE Syntax

```
CLOSE { file-name-1 [ { REEL|UNIT [ FOR REMOVAL ] } ] }...
~~~~~                 { ~~~~ ~~~~       ~~~~~~~   }
                      { WITH LOCK                 }
                      {      ~~~~                 }
                      { WITH NO REWIND            }
                             ~~ ~~~~~~
```

The `REEL`, `LOCK` and `NO REWIND` clauses are syntactically recognized but are otherwise non-functional, except for the `CLOSE...NO REWIND` statement, which will generate a file status of 07 rather than the usual 00 (but take no other action).

The `CLOSE` statement terminates the program's access to the specified file(s).

1. The reserved words `FOR` and `WITH` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved words `REEL` and `UNIT` are interchangeable.
3. The `CLOSE` statement may only be executed against files that have been successfully opened.
4. A successful `CLOSE` will write any remaining unwritten record buffers to the file (similar to an `UNLOCK` statement (see [UNLOCK], page 412)) and release any file locks for the file, regardless of open mode. A closed file will then be no longer available for subsequent I/O statements until it is once again OPENED.
5. When a `ORGANIZATION LINE SEQUENTIAL` (see [ORGANIZATION LINE SEQUENTIAL], page 111) or `LINE ADVANCING` (see [LINE ADVANCING], page 15) file is closed, a final delimiter sequence will be written to the file to signal the termination point of the final data record in the file. This will only be necessary if the final record written to the file was written with the `AFTER ADVANCING` (see [WRITE], page 417) option.

7.8.8. COMMIT

COMMIT Syntax

```
COMMIT
~~~~~~
```

The `COMMIT` statement performs an `UNLOCK` against every currently-open file, but does not close any of the files. See the `UNLOCK` statement (see [UNLOCK], page 412) for additional details.

7.8.9. COMPUTE

COMPUTE Syntax

```
COMPUTE { identifier-1
~~~~~~~
    [ ROUNDED [ MODE IS { AWAY-FROM-ZERO         } ] }...
      ~~~~~~~   ~~~~    { ~~~~~~~~~~~~~~         }
                        { NEAREST-AWAY-FROM-ZERO }
                        { ~~~~~~~~~~~~~~~~~~~~~~ }
                        { NEAREST-EVEN           }
                        { ~~~~~~~~~~~~           }
                        { NEAREST-TOWARD-ZERO    }
                        { ~~~~~~~~~~~~~~~~~~     }
                        { PROHIBITED             }
                        { ~~~~~~~~~~             }
                        { TOWARD-GREATER         }
                        { ~~~~~~~~~~~~~~         }
                        { TOWARD-LESSER          }
                        { ~~~~~~~~~~~~~          }
                        { TRUNCATION             }
                          ~~~~~~~~~~

      =|EQUAL arithmetic-expression-1
        ~~~~~

    [ ON SIZE ERROR imperative-statement-1 ]
        ~~~~ ~~~~~

    [ NOT ON SIZE ERROR imperative-statement-2 ]
      ~~~    ~~~~ ~~~~~

[ END-COMPUTE ]
  ~~~~~~~~~~~
```

The `COMPUTE` statement provides a means of easily performing complex arithmetic operations with a single statement, instead of using cumbersome and possibly confusing sequences of `ADD`, `SUBTRACT`, `MULTIPLY` and `DIVIDE` statements. `COMPUTE` also allows the use of exponentiation — an arithmetic operation for which no other statement exists in COBOL.

1. The reserved words `IS` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved word `EQUAL` is interchangeable with the use of '`=`'.
3. Each *identifier-1* must be a numeric or numeric-edited data item.
4. The optional `ROUNDED` (see [ROUNDED], page 261) clause available to each *identifier-1* will control how non-integer results will be saved.
5. See [Arithmetic Expressions], page 42, for more information on arithmetic expressions.

6. The optional `ON SIZE ERROR` and `NOT ON SIZE ERROR` clauses may be used to detect and react to the failure or success, respectively, of an attempt to perform a calculation. In this case, failure is defined either as having an *identifier-3* with an insufficient number of digit positions available to the left of any implied decimal point or attempting to divide by zero. See [ON SIZE ERROR + NOT ON SIZE ERROR], page 261, for additional information.

7.8.10. CONTINUE

CONTINUE Syntax

```
CONTINUE
~~~~~~~~

                { identifier-1            }
CONTINUE AFTER  { literal-1               }   SECONDS
~~~~~~~~ ~~~~~  { arithmetic-expression-1 }   ~~~~~~~
```

The `CONTINUE` statement is a no-operation statement that may be coded anywhere an imperative statement (see [Imperative Statement], page 722) may be coded.

1. The `CONTINUE` statement has no effect on the execution of the program.
2. This statement (perhaps in combination with an appropriate comment or two) makes a convenient "place holder" — particularly in `ELSE` (see [IF], page 338) or `WHEN` (see [EVALUATE], page 320) clauses where no code is currently expected to be needed, but a place for code to handle the conditions in question is to be reserved in case it's ever needed.
3. The optional extension of (AFTER) when used with the `CONTINUE` statement pauses execution for a specified length of time.

7.8.11. DELETE

DELETE Syntax

```
DELETE file-name-1 RECORD
~~~~~~
  [ INVALID KEY imperative-statement-1 ]
    ~~~~~~~
  [ NOT INVALID KEY imperative-statement-2 ]
    ~~~ ~~~~~~~
[ END-DELETE ]
  ~~~~~~~~~~
```

The `DELETE` statement logically deletes a record from a COBOL file.

1. The reserved words `KEY` and `RECORD` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The `ORGANIZATION` of *file-name-1* cannot be `ORGANIZATION LINE SEQUENTIAL` (see [ORGANIZATION LINE SEQUENTIAL], page 111).
3. The *file-name-1* file cannot be a sort/merge work file (a file described using a `SD` (see [File/Sort-Description], page 124)).
4. For files in the `SEQUENTIAL` access mode, the last input-output statement executed against *file-name-1* prior to the execution of the `DELETE` statement must have been a successfully executed sequential-format `READ` statement (see [Sequential READ], page 366). That `READ` will therefore identify the record to be deleted.
5. If *file-name-1* is a `RELATIVE` file whose `ACCESS MODE` (see [ORGANIZATION RELATIVE], page 113) is either `RANDOM` or `DYNAMIC`, the record to be deleted is the one whose relative record number is currently the value of the field specified as the files `RELATIVE KEY` in its `SELECT` statement.
6. If *file-name-1* is an `INDEXED` file whose `ACCESS MODE` (see [ORGANIZATION INDEXED], page 115) is `RANDOM` or `DYNAMIC`, the record to be deleted is the one whose primary key is currently the value of the field specified as the `RECORD KEY` in the file's `SELECT` statement.
7. The optional `INVALID KEY` and `NOT INVALID KEY` clauses may be used to detect and react to the failure or success, respectively, of an attempt to delete a record. See [INVALID KEY + NOT INVALID KEY], page 259, for additional information.
8. No `INVALID KEY` or `NOT INVALID KEY` clause may be specified for a file who's `ACCESS MODE IS SEQUENTIAL`.

7.8.12. DISPLAY

7.8.12.1. DISPLAY UPON device

DISPLAY UPON device Syntax

```
DISPLAY { literal-1    }...
~~~~~~~ { identifier-1 }
   [ UPON mnemonic-name-1 ]
     ~~~~
   [ WITH NO ADVANCING ]
         ~~ ~~~~~~~~~
   [ ON EXCEPTION imperative-statement-1 ]
       ~~~~~~~~~
   [ NOT ON EXCEPTION imperative-statement-2 ]
     ~~~    ~~~~~~~~~
[ END-DISPLAY ]
  ~~~~~~~~~~~
```

This format of the `DISPLAY` statement displays the specified identifier contents and/or literal values on the system output device specified via the `UPON` clause.

1. The reserved words `ON` and `WITH` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. If no `UPON` clause is specified, `UPON CONSOLE` will be assumed. If the `UPON` clause *is* specified, *mnemonic-name-1* must be one of the built-in output device names `CONSOLE`, `PRINTER`, `STDERR`, `STDOUT`, `SYSERR`, `SYSLIST`, `SYSLST` or `SYSOUT` or a mnemonic name assigned to one of those devices via the `SPECIAL-NAMES` (see [SPECIAL-NAMES], page 92) paragraph.

   When displaying upon the `STDERR` or `SYSERR` devices or to a *mnemonic-name-1* attached to one of those two devices, the output will be written to output pipe #2, which will normally cause the output to appear in the console output window. You may, if desired, redirect that output to a file by appending *`2> filename`* to the end of the command that executes the program. This applies to both Windows (any type) or Unix versions of GnuCOBOL.

   When displaying upon the `CONSOLE`, `PRINTER`, `STDOUT`, `SYSLIST`, `SYSLST` or `SYSOUT` devices or to a *mnemonic-name-1* attached to one of them, the output will be written to output pipe #1, which will normally cause the output to appear in the console output window. You may, if desired, redirect that output to a file by appending *`1> filename`* or simply *`> filename`* to the end of the command that executes the program. This applies to both Windows (any type) or Unix versions of GnuCOBOL.
3. The `NO ADVANCING` clause, if used, will suppress the carriage-return / line-feed sequence that is normally added to the end of any console display.

4. The optional `ON EXCEPTION` and `NOT ON EXCEPTION` clauses may be used to detect and react to the failure or success, respectively, of an attempt to display output to the specified device. See [ON EXCEPTION + NOT ON EXCEPTION], page 260, for additional information.

7.8.12.2. DISPLAY UPON COMMAND-LINE

DISPLAY UPON COMMAND-LINE Syntax

```
DISPLAY { literal-1    }...
~~~~~~~ { identifier-1 }
     UPON { ARGUMENT-NUMBER|COMMAND-LINE }
     ~~~~ { ~~~~~~~~~~~~~~~ ~~~~~~~~~~~~ }
   [ ON EXCEPTION imperative-statement-1 ]
        ~~~~~~~~~
   [ NOT ON EXCEPTION imperative-statement-2 ]
     ~~~    ~~~~~~~~~
[ END-DISPLAY ]
  ~~~~~~~~~~~
```

This form of the `DISPLAY` statement may be used to specify the command-line argument number to be retrieved by a subsequent `ACCEPT FROM ARGUMENT-VALUE` statement (see [ACCEPT FROM COMMAND-LINE], page 269) or to specify a new value for the command-line arguments themselves.

1. The reserved word `ON` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. By displaying a numeric integer value `UPON ARGUMENT-NUMBER`, you will specify which argument (by its relative number) will be retrieved by a subsequent `ACCEPT FROM ARGUMENT-VALUE` statement.
3. Executing a `DISPLAY UPON COMMAND-LINE` will influence subsequent `ACCEPT FROM COMMAND-LINE` statements (which will then return the value you displayed), but will not influence subsequent `ACCEPT FROM ARGUMENT-VALUE` statements — these will continue to return the original program execution parameters.
4. The optional `ON EXCEPTION` and `NOT ON EXCEPTION` clauses may be used to detect and react to the failure or success, respectively, of an attempt to display output to the specified item. See [ON EXCEPTION + NOT ON EXCEPTION], page 260, for additional information.

7.8.12.3. DISPLAY UPON ENVIRONMENT-NAME

DISPLAY UPON ENVIRONMENT-NAME Syntax

```
DISPLAY { literal-1    }...  UPON { ENVIRONMENT-VALUE }
        { identifier-1 }           { ~~~~~~~~~~~~~~~~~ }
                                   { ENVIRONMENT-NAME  }
                                     ~~~~~~~~~~~~~~~~

   [ ON EXCEPTION imperative-statement-1 ]
        ~~~~~~~~~

   [ NOT ON EXCEPTION imperative-statement-2 ]
     ~~~    ~~~~~~~~~

[ END-DISPLAY ]
  ~~~~~~~~~~~
```

This form of the `DISPLAY` statement can be used to create or modify environment variables.

1. The reserved word `ON` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. To create or change an environment variable will require two `DISPLAY` statements. The following example sets the environment variable `MY_ENV_VAR` to a value of '`Demonstration Value`':

   ```
   DISPLAY "MY_ENV_VAR" UPON ENVIRONMENT-NAME
   DISPLAY "Demonstration Value" UPON ENVIRONMENT-VALUE
   ```

3. Environment variables created or changed from within GnuCOBOL programs will be available to any sub-shell processes spawned by that program (i.e. `CALL 'SYSTEM'` (see [SYSTEM], page 599)) but will not be known to the shell or console window that started the GnuCOBOL program.
4. Consider using `SET ENVIRONMENT` (see [SET ENVIRONMENT], page 382) in lieu of `DISPLAY` to set environment variables as it is much simpler.
5. The optional `ON EXCEPTION` and `NOT ON EXCEPTION` clauses may be used to detect and react to the failure or success, respectively, of an attempt to display output to the specified item. See [ON EXCEPTION + NOT ON EXCEPTION], page 260, for additional information.

7.8.12.4. DISPLAY Data-Item

DISPLAY Data-Item Syntax

```
DISPLAY identifier-1 [ UPON CRT|CRT-UNDER ]
~~~~~~~                ~~~~ ~~~ ~~~~~~~~~

        OMITTED
        ~~~~~~~

    [ AT { | LINE NUMBER { integer-1    }            | } ]
      ~~ { | ~~~~        { identifier-2 }            | }
         { | COLUMN|COL|POSITION|POS NUMBER { integer-2    }
         { | ~~~~~~ ~~~ ~~~~~~~~ ~~~        { identifier-3 }
         {                                              }
         { { integer-3    }                             }
         { { identifier-4 }                             }

    [ WITH [ Attribute-Specification ]...
      ~~~~
           [ SCROLL { UP   } [ { integer-4    } LINE|LINES ] ]
             ~~~~~~ { ~~   }   { identifier-5 }
                    { DOWN }
                      ~~~~
           [ SIZE { integer-5    }
             ~~~~ { identifier-6 }  ]
           [ NO ADVANCING ]
             ~~~~~~~~~~~~

    [ ON EXCEPTION imperative-statement-1 ]
       ~~~~~~~~~

    [ NOT ON EXCEPTION imperative-statement-2 ]
      ~~~    ~~~~~~~~~

[ END-DISPLAY ]
  ~~~~~~~~~~~
```

The `UPON CRT`, `UPON CRT-UNDER` and `CONVERSION` clauses are syntactically recognized but are otherwise non-functional. They are supported to provide compatibility with COBOL source written for other COBOL implementations.

This format of the `DISPLAY` statement presents data onto a formatted screen.

1. The reserved words `AFTER`, `LINE`, `LINES`, `NUMBER` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved words `COLUMN` and `POSITION` are interchangeable.
3. The reserved words `LINE` and `LINES` are interchangeable.
4. If *identifier-1* is defined in the `SCREEN SECTION` (see [SCREEN SECTION], page 143), any `AT`, *Attribute-Specification* and `WITH` clauses will be ignored. All field definition,

cursor positioning and screen control will occur as a result of the screen section definition of *identifier-1*.

5. The reserved word `OMITTED` when used, will act to position the cursor or any screen clearance without changing any content of the screen.
6. The following points apply if *identifier-1* is not defined in the screen section:
   A. The purpose of the `AT` clause is to define where on the screen *identifier-1* should be displayed. See [ACCEPT data-item], page 272, for additional information.
   B. The purpose of the `WITH` clause is to define the visual attributes that should be applied to *identifier-1* when it is displayed on the screen as well as other presentation-control characteristics.
   C. The following *Attribute-Specification* clauses are allowed on the `DISPLAY` statement — these are the same as those allowed for `SCREEN SECTION` data items. A particular *Attribute-Specification* may be used only once in any `DISPLAY`:
      - `BACKGROUND-COLOR` (see [BACKGROUND-COLOR], page 158)
      - `BEEP` (see [BEEP], page 161), `BELL` (see [BELL], page 162)
      - `BLANK` (see [BLANK], page 163)
      - `BLINK` (see [BLINK], page 165)
      - `ERASE` (see [ERASE], page 170)
      - `FOREGROUND-COLOR` (see [FOREGROUND-COLOR], page 173)
      - `HIGHLIGHT` (see [HIGHLIGHT], page 178)
      - `LOWLIGHT` (see [LOWLIGHT], page 186)
      - `OVERLINE` (see [OVERLINE], page 193)
      - `REVERSE-VIDEO` (see [REVERSE-VIDEO], page 207)
      - `UNDERLINE` (see [UNDERLINE], page 229)
   D. See [ACCEPT data-item], page 272, for additional information on the other `WITH` clause options.
7. The optional `ON EXCEPTION` and `NOT ON EXCEPTION` clauses may be used to detect and react to the failure or success, respectively, of the screen I/O attempt. See [ON EXCEPTION + NOT ON EXCEPTION], page 260, for additional information.

When `DISPLAY` is used with Line and column where multiple variables or literals are used before `LINE` only the first will be displayed.

If this is needed then the use of `CONCATENATE` to built more than one element together prior to the display, *e.g.*, `DISPLAY FUNCTION CONCATENATE (VARS-1 VARS-2) AT 0201`.

When `DISPLAY` is used without line or column controls only one variable or literal may will appear on a line, so the use of the above example should also be employed.

7.8.12.5. DISPLAY data-item

DISPLAY data-item Syntax

```
DISPLAY  [position-spec] {identifier-2 | literal-1} ...
~~~~~~~

  [ WITH [ Attribute-Specification ]...
    ~~~~
         [ ERASE  { SCREEN|LINE }  ]
         [ SCROLL { UP   } [ { integer-3    } LINE|LINES ] ]
           ~~~~~~  { ~~   }   { identifier-3 }
                                { DOWN }
                                  ~~~~
         [ SIZE { integer-4    }
           ~~~~ { identifier-4 } ]
[ END-DISPLAY ]
  ~~~~~~~~~~~


  where  position-spec is
  { (position-spec-num, position-spec-num) }
  { (,position-spec-num)                   }
  { (position-spec-num,)                   }

  where  position-spec-num  is
  { identifier-1 } [{ + } integer-1 ]
  { integer-2    } [{ - }           ]
```

This format of the `DISPLAY` statement presents data onto a formatted screen using the Microsoft format from v1 and v2 compilers (MsDos).

7.8.13. DIVIDE

7.8.13.1. DIVIDE INTO

DIVIDE INTO Syntax

```
DIVIDE { literal-1    } INTO { literal-2    } GIVING { identifier-3
~~~~~~ { identifier-1 } ~~~~ { identifier-2 } ~~~~~~

           [ ROUNDED [ MODE IS { AWAY-FROM-ZERO         } ] ] }...
             ~~~~~~~   ~~~~    { ~~~~~~~~~~~~~~         }
                               { NEAREST-AWAY-FROM-ZERO }
                               { ~~~~~~~~~~~~~~~~~~~~~~ }
                               { NEAREST-EVEN           }
                               { ~~~~~~~~~~~~           }
                               { NEAREST-TOWARD-ZERO    }
                               { ~~~~~~~~~~~~~~~~~~     }
                               { PROHIBITED             }
                               { ~~~~~~~~~~             }
                               { TOWARD-GREATER         }
                               { ~~~~~~~~~~~~~~         }
                               { TOWARD-LESSER          }
                               { ~~~~~~~~~~~~~          }
                               { TRUNCATION             }
                                 ~~~~~~~~~~

    [ REMAINDER identifier-4 ]
      ~~~~~~~~~

    [ ON SIZE ERROR imperative-statement-1 ]
         ~~~~ ~~~~~

    [ NOT ON SIZE ERROR imperative-statement-2 ]
      ~~~    ~~~~ ~~~~~

 [ END-DIVIDE ]
   ~~~~~~~~~~
```

For further clarification, the following examples are provided to be used with the various flavours of the DIVIDE statement when using BY, INTO and GIVING.

DIVIDE Operation	A	B	C	D
DIVIDE A INTO B	A	B/A		
DIVIDE A INTO B GIVING C	A	B	B/A	
DIVIDE A BY B GIVING C	A	B	A/B	

```
DIVIDE A INTO B GIVING C REMAINDER D  | A |  B  | Integer(B/A) | Integer remainder |
--------------------------------------+---+-----+--------------+-------------------+
```

This format of the `DIVIDE` statement will divide a numeric value (specified as a literal or numeric data item) into another numeric value (also specified as a literal or numeric data item) and will then replace the contents of one or more receiving data items with the results of that division.

1. The reserved words `IS` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. Both *identifier-1* and *identifier-2* must be numeric unedited data items and *literal-1* must be a numeric literal.
3. A division operation will be performed for each *identifier-2*, in turn. Each of the results of those divisions will be saved to the corresponding *identifier-2* data item(s).
4. Should any *identifier-2* be an integer numeric data item, the result computed when that *identifier-2* is divided by *literal-1* or *identifier-1* will also be an integer — any remainder from that division will be discarded.
5. The optional `ROUNDED` (see [ROUNDED], page 261) clause available to each *identifier-2* will control how non-integer results will be saved.
6. The optional `ON SIZE ERROR` and `NOT ON SIZE ERROR` clauses may be used to detect and react to the failure or success, respectively, of an attempt to perform a calculation. In this case, failure is defined as being numeric truncation caused by an *identifier-2* with an insufficient number of digit positions available to the left of any implied decimal point, or an attempt to divide by zero. See [ON SIZE ERROR + NOT ON SIZE ERROR], page 261, for additional information.

7.8.13.2. DIVIDE INTO GIVING

DIVIDE INTO GIVING Syntax

```
DIVIDE { literal-1    } INTO { literal-2    } GIVING { identifier-3
~~~~~~ { identifier-1 } ~~~~ { identifier-2 } ~~~~~~

           [ ROUNDED [ MODE IS { AWAY-FROM-ZERO         } ] ] }...
             ~~~~~~~   ~~~~    { ~~~~~~~~~~~~~~         }
                               { NEAREST-AWAY-FROM-ZERO }
                               { ~~~~~~~~~~~~~~~~~~~~~~ }
                               { NEAREST-EVEN           }
                               { ~~~~~~~~~~~~           }
                               { NEAREST-TOWARD-ZERO    }
                               { ~~~~~~~~~~~~~~~~~~     }
                               { PROHIBITED             }
                               { ~~~~~~~~~~             }
                               { TOWARD-GREATER         }
                               { ~~~~~~~~~~~~~~         }
                               { TOWARD-LESSER          }
                               { ~~~~~~~~~~~~           }
                               { TRUNCATION             }
                                 ~~~~~~~~~~

   [ REMAINDER identifier-4 ]
     ~~~~~~~~~

   [ ON SIZE ERROR imperative-statement-1 ]
        ~~~~ ~~~~~

   [ NOT ON SIZE ERROR imperative-statement-2 ]
     ~~~    ~~~~ ~~~~~

[ END-DIVIDE ]
  ~~~~~~~~~~
```

This format of the `DIVIDE` statement will divide one numeric value (specified as a literal or numeric data item) into another numeric value (also specified as a literal or numeric data item) and will then replace the contents of one or more receiving data items with the results of that division.

1. The reserved words `IS` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. Both *identifier-1* and *identifier-2* must be numeric unedited data items while both *identifier-3* and *identifier-4* must be numeric (edited or unedited) data items.
3. Both *literal-1* and *literal-2* must be numeric literals.
4. If the `REMAINDER` clause is coded, there may be only one *identifier-3* specified.

5. The result obtained when the value of *literal-2* or *identifier-2* is divided by the value of *literal-1* or *identifier-1* is computed; this result is then moved into each *identifier-3*, in turn, applying the rules defined by the `ROUNDED` (see [ROUNDED], page 261) clause (if any) for that *identifier-3* to the move.
6. If a `REMAINDER` clause is specified, the value of the one and only *identifier-3* (as stated earlier, if `REMAINDER` is specified there may only be a single *identifier-3* coded on the statement) after it was assigned a value according to the previous rule will be multiplied by the value of *literal-1* or *identifier-1*; that result is then subtracted from the value of *literal-2* or *identifier-2* and *that* result is the value which is moved to *identifier-4*.
7. The optional `ON SIZE ERROR` and `NOT ON SIZE ERROR` clauses may be used to detect and react to the failure or success, respectively, of an attempt to perform a calculation. In this case, failure is defined as being an *identifier-2* with an insufficient number of digit positions available to the left of any implied decimal point, or an attempt to divide by zero. See [ON SIZE ERROR + NOT ON SIZE ERROR], page 261, for additional information.

7.8.13.3. DIVIDE BY GIVING

DIVIDE BY GIVING Syntax

```
DIVIDE { literal-1    } BY { literal-2    } GIVING { identifier-3
~~~~~~ { identifier-1 } ~~ { identifier-2 } ~~~~~~

           [ ROUNDED [ MODE IS { AWAY-FROM-ZERO         } ] ] }...
             ~~~~~~~   ~~~~    { ~~~~~~~~~~~~~~         }
                               { NEAREST-AWAY-FROM-ZERO }
                               { ~~~~~~~~~~~~~~~~~~~~~~ }
                               { NEAREST-EVEN           }
                               { ~~~~~~~~~~~~           }
                               { NEAREST-TOWARD-ZERO    }
                               { ~~~~~~~~~~~~~~~~~~     }
                               { PROHIBITED             }
                               { ~~~~~~~~~~             }
                               { TOWARD-GREATER         }
                               { ~~~~~~~~~~~~~~         }
                               { TOWARD-LESSER          }
                               { ~~~~~~~~~~~~           }
                               { TRUNCATION             }
                                 ~~~~~~~~~~

   [ REMAINDER identifier-4 ]
     ~~~~~~~~~

   [ ON SIZE ERROR imperative-statement-1 ]
        ~~~~ ~~~~~

   [ NOT ON SIZE ERROR imperative-statement-2 ]
     ~~~    ~~~~ ~~~~~

[ END-DIVIDE ]
  ~~~~~~~~~~
```

This format of the `DIVIDE` statement will divide one numeric value (specified as a literal or numeric data item) by another numeric value (also specified as a literal or numeric data item) and will then replace the contents of one or more receiving data items with the results of that division.

1. The reserved words `IS` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. Both *identifier-1* and *identifier-2* must be numeric unedited data items while both *identifier-3* and *identifier-4* must be numeric (edited or unedited) data items.
3. Both *literal-1* and *literal-2* must be numeric literals.
4. If the `REMAINDER` clause is coded, there may be only one *identifier-3* specified.

5. The result obtained when the value of *literal-1* or *identifier-1* is divided by the value of *literal-2* or *identifier-2* is computed; this result is then moved into each *identifier-3*, in turn, applying the rules defined by the `ROUNDED` (see [ROUNDED], page 261) clause (if any) for that *identifier-3* to the move.
6. If a `REMAINDER` clause is specified, the value of the one and only *identifier-3* (as stated earlier, if `REMAINDER` is specified there may only be a single *identifier-3* coded on the statement) after it was assigned a value according to the previous rule will be multiplied by the value of *literal-2* or *identifier-2*; that result is then subtracted from the value of *literal-1* or *identifier-1* and *that* result is the value which is moved to *identifier-4*.
7. The optional `ON SIZE ERROR` and `NOT ON SIZE ERROR` clauses may be used to detect and react to the failure or success, respectively, of an attempt to perform a calculation. In this case, failure is defined as being an *identifier-2* with an insufficient number of digit positions available to the left of any implied decimal point, or an attempt to divide by zero. See [ON SIZE ERROR + NOT ON SIZE ERROR], page 261, for additional information.

7.8.14. ENTRY

ENTRY Syntax

```
ENTRY [ call-convention-phrase ] literal-1
~~~~~
    [ linkage-phrase ]

 [ USING ENTRY-Argument ...  ]
   ~~~~~

Format 2 (Special purpose and for GO TO )

ENTRY FOR GO TO literal-3
~~~~~ ~~~ ~~ ~~
```

ENTRY-Argument Syntax

```
[ BY { REFERENCE } ] { OMITTED }
     { ~~~~~~~~~ }
     { CONTENT   } ] { [ size-phrase ] { identifier-1 } }
     { ~~~~~~~   }                     { literal-2    } }
     { VALUE     } ] { [ size-phrase ] { identifier-1 } }
       ~~~~~                           { literal-2    } }
```

The `ENTRY` statement is used to define an alternate entry-point into a subroutine, along with the arguments that subroutine will be expecting.

1. The reserved word `BY` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. You may not use an `ENTRY` statement in a nested subprogram, nor may you use it in any form of user-defined function.
3. The `USING` clause defines the arguments the subroutine entry-point supports. This list of arguments must match up against the `USING` clause of any `CALL` statement that will be invoking the subroutine using this entry-point.
4. Each *ENTRY-Argument* specified on the `ENTRY` statement must be defined in the linkage section of the subroutine in which the `ENTRY` statement exists.
5. The *literal-1* value will specify the entry-point name of the subroutine. It must be specified exactly on `CALL` statements (with regard to the use of upper- and lower-case letters) as it is specified on the `ENTRY` statement.
6. The meaning of `REFERENCE`, `CONTENT` and `VALUE` are the same as the equivalent specifications on the `CALL` statement (see [CALL], page 293). Whatever specification will

be used for an argument on the `CALL` to this entry-point should match the specification used in the corresponding *ENTRY-Argument*. The same rules regarding the presence or absence of a `BY` clause on a `CALL` statement apply to the presence or absence of a `BY` clause on the corresponding argument of the `ENTRY` statement.

7. The *GO TO* with the ENTRY FOR is an GnuCOBOL special purpose extension for use with various GnuCobol tools.

7.8.15. EVALUATE

EVALUATE Syntax

```
  EVALUATE Selection-Subject-1 [ ALSO Selection-Subject-2 ]...
  ~~~~~~~~                       ~~~~
{ { WHEN Selection-Object-1 [ ALSO Selection-Object-2 ] }...
    ~~~~                     ~~~~
        [ imperative-statement-1 ] }...
  [ WHEN OTHER
    ~~~~ ~~~~~
        imperative-statement-other ]

[ END-EVALUATE ]
  ~~~~~~~~~~~~
```

EVALUATE Selection Subject Syntax

```
{ TRUE         }
{ ~~~~         }
{ FALSE        }
{ ~~~~~        }
{ expression-1 }
{ identifier-1 }
{ literal-1    }
```

EVALUATE Selection Object Syntax

```
{ ANY                                                 }
{ ~~~                                                 }
{ TRUE                                                }
{ ~~~~                                                }
{ FALSE                                               }
{ ~~~~~                                               }
{ partial-expression-1                                }
{                                                     }
{ { expression-2 } [ THRU|THROUGH { expression-3 } ] }
{ { identifier-2 }   ~~~~ ~~~~~~~ { identifier-3 }   }
{ { literal-2    }                { literal-3    }   }
```

The EVALUATE statement provides a means of defining processing that should take place under any number of mutually-exclusive conditions.

1. The reserved words THRU and THROUGH are interchangeable.
2. There must be at least one WHEN clause (in addition to any WHEN OTHER clause) specified on any EVALUATE statement.
3. There must be at least one *Selection-Subject* specified on the EVALUATE statement. Any number of additional *Selection-Subject* clauses may be specified, using the ALSO reserved word to separate each from the prior.
4. Each WHEN clause (other than the WHEN OTHER clause, if any) must have the same number of *Selection-Object* clauses as there are *Selection-Subject* clauses.
5. When using THRU, the values on both sides of the THRU must be the same class (both numeric, both alphanumeric, etc.).
6. A *partial-expression* is one of the following:
   A. A Class Condition without a leading *identifier-1* (see [Class Conditions], page 46).
   B. A Sign Condition without a leading *identifier-1* (see [Sign Conditions], page 48).
   C. A Relation Condition with nothing to the left of the relational operator (see [Relation Conditions], page 50).
7. At execution time, each *Selection-Subject* on the EVALUATE statement will have its value matched against that of the corresponding *Selection-Object* on a WHEN clause, in turn, until:
   A. A WHEN clause has *each* of its *Selection-Object*(s) successfully matched by the corresponding *Selection-Subject*; this will be referred to as the '*Selected WHEN clause*'.
   B. The complete list of WHEN clauses (except for the WHEN OTHER clause, if any) has been exhausted. In this case, there is no '*Selected WHEN Clause*'.
8. If a '*Selected WHEN Clause*' was identified:
   A. The *imperative-statement-1* (see [Imperative Statement], page 722) immediately following the '*Selected WHEN Clause*' will be executed. If the '*Selected WHEN Clause*' is lacking an *imperative-statement-1*, the first *imperative-statement-1* found after any following WHEN clause will be executed.
   B. Once the *imperative-statement-1* has been executed, or no *imperative-statement-1* was found anywhere after the '*Selected WHEN Clause*', control will proceed to the statement following the END-EVALUATE or, if there is no END-EVALUATE, the first statement that follows the next period. If, however, the *imperative-statement-1* included a GO TO statement, and that GO TO was executed, then control will transfer to the procedure named on the GO TO instead.
9. If no '*Selected WHEN Clause*' was identified:
   A. The WHEN OTHER clause's *imperative-statement-other* will be executed, if such a clause was coded.
   B. Control will then proceed to the statement following the END-EVALUATE or the first statement that follows the next period if there is no END-EVALUATE. If,however, the *imperative-statement-other* included a GO TO statement, and that GO TO was executed, then control will transfer to the procedure named on the GO TO instead.

10. In order for a *Selection-Subject* to match the corresponding *Selection-Object* on a `WHEN` clause, at least one of the following must be true:
    A. The *Selection-Object* is `ANY`
    B. The implied Relation Condition *`Selection-Subject = Selection Object`* is `TRUE` — See [Relation Conditions], page 50, for the rules on how the comparison will be made.
    C. The value of the *Selection-Subject* falls within the range of values specified by the `THRU` clause of the *Selection-Object*
    D. If the *Selection-Object* is a *partial-expression*, then the conditional expression that would be represented by coding *`Selection-Subject Selection-Object`* evaluates to `TRUE`
11. Here is a sample program that illustrates the EVALUATE statement.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DEMOEVALUATE.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Test-Digit                  PIC 9(1).
    88 Digit-Is-Odd VALUE 1, 3, 5, 7, 9.
    88 Digit-Is-Prime VALUE 1, 3, 5, 7.
PROCEDURE DIVISION.
P1. PERFORM UNTIL EXIT
    DISPLAY "Enter a digit (0 Quits): "
        WITH NO ADVANCING
    ACCEPT Test-Digit
    IF Test-Digit = 0
        EXIT PERFORM
    END-IF
    EVALUATE Digit-Is-Odd ALSO Digit-Is-Prime
    WHEN TRUE ALSO FALSE
        DISPLAY Test-Digit " is ODD"
            WITH NO ADVANCING
    WHEN TRUE ALSO TRUE
        DISPLAY Test-Digit " is PRIME"
            WITH NO ADVANCING
    WHEN FALSE ALSO ANY
        DISPLAY Test-Digit " is EVEN"
            WITH NO ADVANCING
    END-EVALUATE
    EVALUATE Test-Digit
    WHEN < 5
        DISPLAY " and it's small too"
    WHEN < 8
        DISPLAY " and it's medium too"
    WHEN OTHER
        DISPLAY " and it's large too"
```

```
    END-EVALUATE
END-PERFORM
DISPLAY "Bye!"
STOP RUN
.
```

Console output when run (user input follows the colons on the prompts for input):

```
Enter a digit (0 Quits): 1
1 is PRIME and it's small too
Enter a digit (0 Quits): 2
2 is EVEN and it's small too
Enter a digit (0 Quits): 3
3 is PRIME and it's small too
Enter a digit (0 Quits): 4
4 is EVEN and it's small too
Enter a digit (0 Quits): 5
5 is PRIME and it's medium too
Enter a digit (0 Quits): 6
6 is EVEN and it's medium too
Enter a digit (0 Quits): 7
7 is PRIME and it's medium too
Enter a digit (0 Quits): 8
8 is EVEN and it's large too
Enter a digit (0 Quits): 9
9 is ODD and it's large too
Enter a digit (0 Quits): 0
Bye!
```

7.8.15B. EXAMINE

EXAMINE Syntax

```
EXAMINE { identifier } TALLYING
~~~~~~~                ~~~~~~~~
    { ALL|LEADING|UNTIL FIRST } literal-1  REPLACING BY literal-2
      ~~~ ~~~~~~~ ~~~~~~~~~~~              ~~~~~~~~~~~~

EXAMINE { identifier } REPLACING { ALL|FIRST|LEADING|UNTIL FIRST }
~~~~~~~                ~~~~~~~~~   ~~~ ~~~~~ ~~~~~~~ ~~~~~~~~~~~

      literal-3 BY literal-4
                ~~
```

The `EXAMINE` statement is the pre runner for INSPECT which should be used over the pre 1970 Cobol standard EXAMINE, and it is used to count the number of times a specified character appears in a data item and/or to replace a character with another character.

1. This statement is only available subject to specific dialects being set when running the GnuCOBOL compiler.
2. In all cases, the description of identifier must be such that its usage is display (explicitly or implicitly).
3. When identifier represents a non numeric data item, examination starts at the leftmost character and proceeds to the right. Each character in the data item is examined in turn. For purposes of the EXAMINE statement, external floating point items are treated as non numeric data items.
4. When identifier represents a numeric data item, this data item must consist of numeric characters, and may possess an operational sign. Examination starts at the leftmost character and proceeds to the right. Each character is examined in turn.
5. If the letter 'S' is used in the PICTURE of the data item description to indicate the presence of an operational sign, the sign is ignored by the EXAMINE statement.
6. Each literal must consist of a single character belonging to a class consistent with that of the identifier; in addition, each literal may be any figurative constant except ALL. If identifier is numeric, each literal must be an unsigned integer or the figurative constant ZERO (ZEROES, ZEROS).
7. When Format 1 is used, an integral count is created which replaces the value of a special register called TALLY, whose implicit description is that of an unsigned integer of five digits.
8. When the ALL option is used, this count represents the number of occurrences of literal-1.
9. When the LEADING option is used, this count represents the number of occurrences of literal-1 prior to encountering a character other than literal-1.
10. When the UNTIL FIRST option is used, this count represents all characters encountered before the first occurrence of literal-1.

11. Whether Format 2 is used, or the REPLACING option of Format 1, the replacement rules are the same. They are as follows:
12. When the ALL option is used, literal-2 is substituted for each occurrence of literal-1.
13. When the LEADING option is used, the substitution of literal-2 for each occurrence of literal-1 terminates as soon as a character other than literal-1 or the right hand boundary of the data item is encountered.
14. When the UNTIL FIRST option is used, the substitution of literal-2 terminates as soon as literal-1 or the right hand boundary of the data item is encountered.

7.8.16. EXHIBIT

EXHIBIT Syntax

```
EXHIBIT [CHANGED][NAMED] [position-spec] [ERASE] {identifier-1 | literal-1} ...
~~~~~~~  ~~~~~~~  ~~~~~                    ~~~~~

        [UPON mnemonic-name-1]
         ~~~~

where position-spec is

  {(position-spec-num, position-spec-num)}
  {(, position-spec-num)                 }
  {(position-spec-num, )                 }

where position-spec-num is

  {identifier-2} [{+} integer-2]
  {integer-1   } [{-}          ]
```

The `EXHIBIT` statement causes an (optionally conditional) display of the literals, and/or identifiers (optionally preceded by the identifier name) specified in the statement for the purposes of debugging.

1. EXHIBIT is only present to ease migrations, this is an archaic language element in GnuCOBOL (but without the archaic message warning because there is no explicit dialect configuration for that other than the reserved word). Depending on the -std used it will either compile or not. As the standard says about archaic language elements:

   it should not be used in new compilation groups because better programming practices exist

   The use of DISPLAY is more portable and allows for the same feature-set (with the exception of CHANGED which GnuCOBOL may not support for a long time and with the need of a literal for NAMED). This statement can be removed from any later version.
2. The reserved words `NAMED`, `CHANGED` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
3. Each identifier specified in the EXHIBIT statement can be any class of data. TALLY and RETURN-CODE are the only special registers that can be used as identifiers.
4. Literals and identifiers displayed by the EXHIBIT statement are separated by a space on the displayed line.
5. Each literal can be any figurative constant other than ALL.
6. If the literal is numeric, it must be an unsigned integer.
7. Each execution of an EXHIBIT NAMED statement displays each identifier or literal specified, with each identifier (including any qualifiers and subscripts) followed by a

"=" (equal sign) and its current value. They all appear on a single line on the order in which they appear in the statement.

8. Each execution of an EXHIBIT CHANGED NAMED statement displays each identifier or literal specified, with each identifier (including any qualifiers and subscripts) followed by a "=" (equal sign) and its current value. They all appear on a single line on the order in which they appear in the statement. However, the display for each identifier (name and value) is conditional on the value of that identifier having changed since the last execution of the current EXHIBIT statement. If one or more of the identifier values have not changed, neither the name nor the value is printed for those identifiers. If none of the identifier values has changed, and no literals are specified, no display takes place (display of a blank line is suppressed).
9. Each execution of an EXHIBIT CHANGED statement displays the current value of each identifier or literal specified. They all appear on a single line on the order in which they appear in the statement. However, the value display for each identifier is conditional on the value of that identifier having changed since the last execution of the current EXHIBIT statement. If one or more of the identifier values have not changed, the value for those identifiers are not printed and spaces are inserted instead. If none of the identifier values has changed, and no literals are specified, a blank line is displayed (display of a blank line is not suppressed).
10. Each execution of an EXHIBIT statement with neither the CHANGED nor the NAMED option displays each identifier or literal specified. They all appear on a single line in the order in which they appear in the statement.
11. An EXHIBIT statement is the same as an EXHIBIT NAMED statement.
12. ERASE is a display attribute which got into GnuCOBOL when MS-COBOL support was increased, it is not yet implemented.
13. The CHANGED clause is not yet implemented but recognised by GnuCOBOL.
14. The UPON mnemonic-name-1 clause is not yet implemented but recognised by GnuCOBOL.

7.8.17. EXIT

EXIT Syntax

```
EXIT [ { PROGRAM           }   [ { RETURNING } ] { identifier-1 } ]
~~~~                           [ { GIVING    } ] { literal-1    } ]
       { FUNCTION          } ]
       { ~~~~~~~~          } ]
       { PERFORM [ CYCLE ] } ]
       { ~~~~~~~   ~~~~~   } ]
       { SECTION           } ]
       { ~~~~~~~           } ]
       { PARAGRAPH         } ]
         ~~~~~~~~~
```

The EXIT statement is a multi-purpose statement; it may provide a common end point for a series of procedures, exit an in-line PERFORM, paragraph or section or it may mark the logical end of a subprogram, returning control back to the calling program.

1. The EXIT PROGRAM statement is not legal anywhere within a user-defined function.
2. The EXIT FUNCTION statement cannot be used anywhere within a subroutine.
3. Neither EXIT PROGRAM nor EXIT FUNCTION may be used within a USE GLOBAL routine in DECLARATIVES (see [DECLARATIVES], page 254).
4. The following points describe the EXIT statement with none of the optional clauses:
   A. When this form of an EXIT statement is used, it must be the only statement in the procedure (paragraph or section) in which it occurs. This is *not* enforced in GnuCOBOL.
   B. This usage of the EXIT statement simply provides a common "GO TO" end point for a series of procedures, as may be seen in the following example:

```
01  Switches.
    05 Input-File-Switch PIC X(1).
       88 EOF-On-Input-File VALUE ''Y'' FALSE ''N''.
...
    SET EOF-On-Input-File TO FALSE.
    PERFORM 100-Process-A-Transaction THRU 199-Exit
        UNTIL EOF-On-Input-File.
...
100-Process-A-Transaction.
    READ Input-File AT END
        SET EOF-On-Input-File TO TRUE
        GO TO 199-Exit
    END-READ.
    IF Input-Rec of Input-File = SPACES
        GO TO 199-Exit  *> IGNORE BLANK RECORDS!
```

```
        END-IF.
        <<process the record just read>>
    199-Exit.
        EXIT.
```

C. In this case, the EXIT statement takes no other run-time action.

5. The following points apply to the EXIT PARAGRAPH and EXIT SECTION statements:
   A. If an EXIT PARAGRAPH statement or EXIT SECTION statement resides in a paragraph *within* the scope of a procedural PERFORM (see [Procedural PERFORM], page 360), control will be returned back to the PERFORM for evaluation of any TIMES, VARYING and/or UNTIL clauses.
   B. If an EXIT PARAGRAPH statement or EXIT SECTION statement resides *outside* the scope of a procedural PERFORM, control simply transfers to the first executable statement in the next paragraph (EXIT PARAGRAPH) or section (EXIT SECTION).
   C. The following shows how the previous example could have been coded without a GO TO by utilizing an EXIT PARAGRAPH statement.

```
01  Switches.
    05 Input-File-Switch PIC X(1).
       88 EOF-On-Input-File VALUE ''Y'' FALSE ''N''.
...
    SET EOF-On-Input-File TO FALSE.
    PERFORM 100-Process-A-Transaction
        UNTIL EOF-On-Input-File.
...
100-Process-A-Transaction.
    READ Input-File AT END
        SET EOF-On-Input-File TO TRUE
        EXIT PARAGRAPH
    END-READ.
    IF Input-Rec of Input-File = SPACES
        EXIT PARAGRAPH *> IGNORE BLANK RECORDS!
    END-IF.
    <<process the record just read>>
```

6. The following points apply to the EXIT PERFORM and EXIT PERFORM CYCLE statements:
   A. The EXIT PERFORM and EXIT PERFORM CYCLE statements are intended to be used in conjunction with an in-line PERFORM statement (see [Inline PERFORM], page 362).
   B. An EXIT PERFORM CYCLE statement will terminate the current iteration of the in-line PERFORM, giving control to any TIMES, VARYING and/or UNTIL clauses for them to determine if another cycle needs to be performed.
   C. An EXIT PERFORM statement will terminate the in-line PERFORM outright, transferring control to the first statement following the END-PERFORM (if there is one) or to the next sentence following the PERFORM if there is no END-PERFORM.
   D. This last example shows the final modification to the previous examples by using an in-line PERFORM along with EXIT PERFORM and EXIT PERFORM CYCLE statements:

```
PERFORM FOREVER
```

```
            READ Input-File AT END
                EXIT PERFORM
            END-READ
            IF Input-Rec of Input-File = SPACES
                EXIT PERFORM CYCLE *> IGNORE BLANK RECORDS!
            END-IF
            <<process the record just read>>
        END PERFORM
```

7. The following points apply to the `EXIT PROGRAM` and `EXIT FUNCTION` statements:
   A. The `EXIT PROGRAM` and `EXIT FUNCTION` statements terminate the execution of a subroutine (i.e. a program that has been CALLed by another) or user-defined function, respectively, returning control back to the calling program.
   B. An `EXIT PROGRAM` statement returns control back to the statement following the `CALL` (see [CALL], page 293) of the subprogram. An `EXIT FUNCTION` statement returns control back to the processing of the statement in the calling program that invoked the user-defined function.
   C. For `EXIT PROGRAM` statement usage of `RETURNING` statement or `GIVING` statement will provide value defined by indentifer-1 or literal-1 back to the calling routine.
   D. If executed by a main program, neither the `EXIT PROGRAM` nor `EXIT FUNCTION` statements will take any action.
   E. The COBOL2002 standard has made a common extension to the COBOL language — the `GOBACK` statement (see [GOBACK], page 334) — a standard language element; the `GOBACK` statement should be strongly considered as the preferred alternative to both `EXIT PROGRAM` and `EXIT FUNCTION` for new subprograms.

7.8.18. FREE

FREE Syntax

```
FREE { [ ADDRESS OF ] identifier-1 }...
~~~~     ~~~~~~~
```

The `FREE` statement releases memory previously allocated to the program by the `ALLOCATE` statement (see [ALLOCATE], page 290).

1. The `ADDRESS OF` clause is optional and may be omitted. The presence or absence of this clause has no effect upon the program.
2. *identifier-1* must have a `USAGE` (see [USAGE], page 232) of `POINTER`, or it must be an 01-level data item with the `BASED` (see [BASED], page 160) attribute.
3. If *identifier-1* is a `USAGE POINTER` data item and it contains a valid address, the `FREE` statement will release the memory block the pointer references. In addition, any `BASED` data items that the pointer was used to provide an address for will become un-based and therefore unusable. If *identifier-1* did not contain a valid address, no action will be taken.
4. If *identifier-1* is a `BASED` data item and that data item is currently based (meaning it currently has memory allocated to it), its memory is released and *identifier-1* will become un-based and therefore unusable. If *identifier-1* was not based, no action will be taken.

7.8.19. GENERATE

GENERATE Syntax

```
GENERATE { report-name-1 }
~~~~~~~~ { identifier-1  }
```

The GENERATE statement presents data to a report.

1. The following points apply when *identifier-1* is specified:
   A. *identifier-1* must be the name of a DETAIL (see [RWCS Lexicon], page 605) report group.
   B. If necessary, *identifier-1* may be qualified with a report name.
   C. The file in whose FD a REPORT clause exists for the report in which *identifier-1* is a detail group must be opened for OUTPUT or EXTEND at the time the GENERATE is executed. See [OPEN], page 358, for information on file open modes.
   D. The report in which *identifier-1* is a DETAIL group must have been successfully initiated via the INITIATE statement (see [INITIATE], page 343) and not yet terminated via the TERMINATE statement (see [TERMINATE], page 410) at the time the GENERATE is executed.
   E. If at least one GENERATE statement of this form is executed against a report, the report is said to be a *detail report*. If no GENERATE statements of this form are executed against a report, the report is said to be a *summary report*.
2. The following points apply when *report-name-1* is specified:
   A. *report-name-1* must be the name of a report having an RD defined for it in the report section.
   B. There must be at least one CONTROL (see [RWCS Lexicon], page 605) group defined for *report-name-1*.
   C. There cannot be more than one DETAIL group defined for *report-name-1*.
   D. The file in whose FD a REPORT *report-name-1* clause exists must be open for OUTPUT or EXTEND at the time the GENERATE is executed.
   E. *report-name-1* must have been successfully initiated (via INITIATE *report-name-1*) and not yet terminated (via TERMINATE) at the time the GENERATE is executed. See [OPEN], page 358, for information on file open modes.
   F. The DETAIL group which is defined for *report-name-1 will* be processed but will not actually be presented to any report page. This will allow summary processing to take place. If all GENERATE statements are of this form, the report is said to be a *summary report*. If at least one GENERATE *identifier-1* is executed, the report is considered to be a *detail report*.
3. When the first GENERATE statement for a report is executed, the contents of all control fields are saved so they may be referenced during the processing of subsequent GENERATE statements.

4. When, during the processing of a subsequent `GENERATE`, it is determined that a control field has changed value (ie. a control break has occurred), the appropriate control footing and control heading processing will take place and a snapshot of the current values of all control fields will again be saved.

7.8.20. GOBACK

GOBACK Syntax

```
GOBACK [ { RETURNING|GIVING { literal-1    }  ]
~~~~~~   { ~~~~~~~~~ ~~~~~~ { identifier-1 }
```

The GOBACK statement is used to logically terminate an executing program.

1. If executed within a subprogram (i.e. a subroutine or user-defined function), GOBACK behaves like an EXIT PROGRAM or EXIT FUNCTION statement, respectively.
2. If executed within a main program, GOBACK will act as a STOP RUN statement.
3. The optional RETURNING clause provides the opportunity to return a numeric value to the operating system (technically, an *exit status* The manner in which the exit status value is interrogated by the operating system varies. Windows can use %ERRORLEVEL% to query the exit status while Unix shells such as sh, bash and ksh can query the exit status as $?. Other Unix shells may have different ways to access the exit status.

7.8.21. GO TO

7.8.21.1. Simple GO TO

Simple GO TO Syntax

```
GO TO procedure-name-1
~~

GO TO ENTRY literal-3
~~    ~~~~~
```

This form of the `GO TO` statement unconditionally transfers control in a program to the first executable statement within the specified *procedure-name-1*.

1. The reserved word `TO` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. If this format of the `GO TO` statement appears in a consecutive sequence of imperative statements (see [Imperative Statement], page 722) within a sentence, it must be the *final* statement in the sentence.
3. If a `GO TO` is executed within the scope of. . .
   A. ...an in-line `PERFORM` (see [PERFORM], page 360), the `PERFORM` is terminated as control of execution transfers to *procedure-name-1*.
   B. ...a procedural `PERFORM` (see [PERFORM], page 360), and *procedure-name-1* lies outside the scope of that `PERFORM`, the `PERFORM` is terminated as control of execution transfers to *procedure-name-1*.
   C. ...a `MERGE` statement (see [MERGE], page 349) `OUTPUT PROCEDURE` or within the scope of either an `INPUT PROCEDURE` or `OUTPUT PROCEDURE` of a `SORT` statement (see [File-Based SORT], page 391), and *procedure-name-1* lies outside the scope of that procedure, the `SORT` or `MERGE` operation is terminated as control of execution transfers to *procedure-name-1*. Any sorted or merged data accumulated to that point is lost.
4. A `GO TO ENTRY` is an GnuCOBOL special purpose extension for use with various GnuCobol tools and is not part of any current ISO standard. See also `ENTRY`.
5. The `GO TO ENTRY` format has to be used together with `ENTRY FOR GO TO`.

7.8.21.2. GO TO DEPENDING ON

GO TO DEPENDING ON Syntax

```
GO TO {procedure-name-1} ...
~~

                              DEPENDING ON identifier-1
                              ~~~~~~~~~~

GO TO ENTRY {literal-3} ...
~~    ~~~~~

                              DEPENDING ON identifier-1
                              ~~~~~~~~~~
```

This form of the `GO TO` statement will transfer control to any one of a number of specified procedure names depending on the numeric value of the identifier specified on the statement.

1. The reserved word `TO` is optional and may be omitted. The presence or absence of this word has no effect upon the program but does help with read-ability.
2. The `PICTURE` (see [PICTURE], page 194) and/or `USAGE` (see [USAGE], page 232) of the specified *identifier-1* must be such as to define it as a numeric, unedited, preferably unsigned integer data item.
3. If the value of *identifier-1* has the value 1, control will be transferred to the 1st specified procedure name. If the value is 2, control will transfer to the 2nd procedure name, and so on.
4. The `GO TO ENTRY ... DEPENDING ON` format has to be used together with `ENTRY FOR GO TO`.

   If control of execution is transferred to a procedure named on the statement, and the `GO TO` is executed within the scope of. . .

   A. ...an in-line `PERFORM` (see [PERFORM], page 360), the `PERFORM` is terminated as control of execution transfers to the procedure named on the statement.

   B. ...a procedural `PERFORM` (see [PERFORM], page 360), and *procedure-name-1* lies outside the scope of that `PERFORM`, the `PERFORM` is terminated as control of execution transfers to the procedure named on the statement.

   C. ...a `MERGE` statement (see [MERGE], page 349) `OUTPUT PROCEDURE` or within the scope of either an `INPUT PROCEDURE` or `OUTPUT PROCEDURE` of a `SORT` statement (see [File-Based SORT], page 391), and *procedure-name-1* lies outside the scope of that procedure, the `SORT` or `MERGE` operation is terminated as control of execution transfers to the procedure named on the statement. Any sorted or merged data accumulated to that point is lost.
5. If the value of *identifier-1* is less than 1 or exceeds the total number of procedure names specified on the statement, control will simply fall through into the next statement following the `GO TO`.

6. The following example shows how `GO TO ... DEPENDING ON` may be used in a real application situation, and compares it against an alternative — `EVALUATE` (see [EVALUATE], page 320).

GO TO DEPENDING ON

```
    GO TO
      ACCT-TYPE-1
      ACCT-TYPE-2
      ACCT-TYPE-3
    DEPENDING ON Acct-Type.
    <<< Invalid Acct Type >>>
    GO TO All-Done.
Acct-Type-1.
    <<< Handle Acct Type 1 >>>
    GO TO All-Done.
Acct-Type-2.
    <<< Handle Acct Type 2 >>>
    GO TO All-Done.
Acct-Type-3.
    <<< Handle Acct Type 3 >>>
All-Done.
```

EVALUATE

```
EVALUATE Acct-Type
WHEN 1
    <<< Handle Acct Type 1 >>>
WHEN 2
    <<< Handle Acct Type 2 >>>
WHEN 3
    <<< Handle Acct Type 3 >>>
WHEN OTHER
    <<< Invalid Acct Type >>>
END-EVALUATE.
```

7. Current programming philosophy would prefer the use of the `EVALUATE` statement to that of this form of the `GO TO` statement.

7.8.22. IF

IF Syntax

```
  IF conditional-expression
  ~~
  THEN { imperative-statement-1 }
       { NEXT SENTENCE          }
         ~~~~ ~~~~~~~~
[ ELSE { imperative-statement-2 } ]
  ~~~~ { NEXT SENTENCE          }
         ~~~~ ~~~~~~~~
[ END-IF ]
  ~~~~~~
```

The IF statement is used to conditionally execute an imperative statement (see [Imperative Statement], page 722) or to select one of two different imperative statements to execute based upon the TRUE/FALSE value of a conditional expression.

1. The reserved word THEN is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. You cannot use both NEXT SENTENCE and the END-IF scope terminator in the same IF statement.
3. If *conditional-expression* evaluates to TRUE, *imperative-statement-1* will be executed regardless of whether or not an ELSE clause is present. Once *imperative-statement-1* has been executed, control falls into the first statement following the END-IF or to the first statement of the next sentence if there is no END-IF clause.
4. If the optional ELSE clause is present and conditional-expression evaluates to false, then (and only then) *imperative-statement-2* will be executed. Once *imperative-statement-2* has been executed, control falls into the first statement following the END-IF or to the first statement of the next sentence if there is no END-IF clause.
5. The clause NEXT SENTENCE may be substituted for either imperative-statement, but not both. If control reaches a NEXT SENTENCE clause due to the truth or falsehood of *conditional-expression*, control will be transferred to the first statement of the next sentence found in the program (the first statement after the next period).

   NEXT SENTENCE was needed for COBOL programs that were coded according to pre-1985 standards that wish to nest one IF statement inside another. See [Use of VERB/END-VERB Constructs], page 55, for an explanation of why NEXT SENTENCE was necessary.

   Programs coded for 1985 (and beyond) standards don't need it, instead using the explicit scope-terminator END-IF to inform the compiler where *imperative-statement-2* (or *imperative-statement-1* if there is no ELSE clause coded) ends. New GnuCOBOL programs should be coded to use the END-IF scope terminator for IF statements. See [Use of VERB/END-VERB Constructs], page 55, for additional information.

7.8.23. INITIALIZE

INITIALIZE Syntax

```
INITIALIZE|INITIALISE identifier-1...
~~~~~~~~~~ ~~~~~~~~~~

    [ WITH FILLER ]
          ~~~~~~

    [ { category-name-1 } TO VALUE ]
      { ALL             }    ~~~~~
        ~~~

    [ THEN REPLACING { category-name-2 DATA BY
          ~~~~~~~~~                       ~~

         [ LENGTH OF ] { literal-1    } }...  ]
           ~~~~~~      { identifier-1 }

    [ THEN TO DEFAULT ]
             ~~~~~~~
```

The `INITIALIZE` statement initializes each *identifier-1* with certain specific values, depending upon the options specified.

1. The reserved words `DATA`, `OF`, `THEN`, `TO` and `WITH` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved words `INITIALIZE` and `INITIALISE` are interchangeable.
3. The `WITH FILLER`, `REPLACING` and `DEFAULT` clauses are meaningful only if *identifier-1* is a group item. They are accepted if it's an elementary item, but will serve no purpose. The `VALUE` clause is meaningful in both cases.
4. A *category-name-1* and/or *category-name-2* may be any of the following:

   ALPHABETIC
   : The `PICTURE` (see [PICTURE], page 194) of the data item only contains `A` symbols.

   ALPHANUMERIC
   : The `PICTURE` of the data item contains only `X` or a combination of `A` and `9` symbols.

   ALPHANUMERIC-EDITED
   : The `PICTURE` of the data item contains only `X` or a combination of `A` and `9` symbols plus at least one `B`, `0` (zero) or `/` symbol.

   NUMERIC
   : The data item is one that is described with a picture less `USAGE` (see [USAGE], page 232) or has a `PICTURE` composed of nothing but `P`, `9`, `S` and `V` symbols.

NUMERIC-EDITED
: The `PICTURE` of the data item contains nothing but the symbol `9` and at least one of the editing symbols `$`, `+`, `-`, `CR`, `DB`, `.`, `,`, `*` or `Z`.

NATIONAL
: The data item is one containing nothing but the `N` symbol.

NATIONAL-EDITED
: The data item contains nothing but `N`, `B`, `/` and `0` symbols.

5. From the sequence of *identifier-1* data items specified on the `INITIALIZE` statement, a list of initialized fields referred to as the *field list* in the remainder of this section, will include:
   A. Every *identifier-1* that is an elementary item, including any that may have the `REDEFINES` (see [REDEFINES], page 204) clause in their descriptions.
   B. Every non-`FILLER` elementary item subordinate to *identifier-1*, provided that elementary item neither contains a `REDEFINES` clause in its definition nor belongs to a group item *subordinate to identifier-1* which contains a `REDEFINES` clause in its definition.
   C. If the optional `WITH FILLER` clause is included on the `INITIALIZE` statement, then every FILLER elementary item subordinate to each *identifier-1* will be included as well, provided that elementary item neither contains a `REDEFINES` clause in its definition nor belongs to a group item *subordinate to identifier-1* which contains a `REDEFINES` clause in its definition..
6. Once a field list has been determined, each item in that field list will be initialized as if an individual `MOVE` (see [MOVE], page 352) statement to that effect had been coded. The rules for initialization are as follows:
7. If no `VALUE`, `REPLACING` or `DEFAULT` clauses are coded, each member of the field list will be initialized as if the figurative constant `ZERO` (if the field list item is numeric or numeric-edited) or `SPACES` (otherwise) were being moved to it.
8. If a `VALUE` clause is specified on the `INITIALIZE` statement, each qualifying member of the field list having a compile-time `VALUE` (see [VALUE], page 244) specified in its definition will be initialized to that value. Field list members with `VALUE` clauses will qualify for this treatment as follows:
   A. If the `ALL` keyword was specified on the `VALUE` clause, all members of the field list with `VALUE` clauses will qualify.
   B. If *category-name-1* is specified instead of `ALL`, only those members of the field list with `VALUE` clauses that also meet the criteria set down for the specified *category-name* (see the list above) will qualify.
   C. If you need to apply `VALUE` initialization to multiple *category-name-1* values, you will need to use multiple `INITIALIZE` statements.
9. If a `REPLACING` clause is specified on the `INITIALIZE` statement, each qualifying member of the field list that was not already initialized by a `VALUE` clause, if any, will be initialized to the specified *literal-1* or *identifier-1* value.

   Only those as-yet uninitialized list members meeting the criteria set forth for the specified *category-name-2* will qualify for this initialization.

If you need to apply REPLACING initialization to multiple *category-name-2* values, you may repeat the syntax after the reserved word REPLACING, as necessary.

10. If a DEFAULT clause is specified, any remaining uninitialized members of the field list will be initialized according to the default for their class (numeric and numeric-edited are initialized to ZERO, all others are initialized to SPACES).

11. The following example may help your understanding of how the INITIALIZE statement works. The sample code makes use of the COBDUMP program to dump the storage that is (or is not) being initialized. See Section "COBDUMP" in *GnuCOBOL Sample Programs*, for a source and cross-reference listing of the COBDUMP program.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DemoInitialize.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Item-1.
    05 I1-A VALUE ALL '*'.
       10 FILLER                PIC X(1).
       10 I1-A-1                PIC 9(1) VALUE 9.
    05 I1-B                     USAGE BINARY-CHAR.
    05 I1-C                     PIC A(1) VALUE 'C'.
    05 I1-D                     PIC X/X VALUE 'ZZ'.
    05 I1-E                     OCCURS 2 TIMES PIC 9.
PROCEDURE DIVISION.
000-Main.
    DISPLAY "MOVE HIGH-VALUES TO Item-1"
        PERFORM 100-Init-Item-1
        CALL "COBDUMP" USING Item-1
        DISPLAY " "

    DISPLAY "INITIALIZE Item-1"
        INITIALIZE Item-1
        CALL "COBDUMP" USING Item-1
        PERFORM 100-Init-Item-1
        DISPLAY " "

    DISPLAY "INITIALIZE Item-1 WITH FILLER"
        MOVE HIGH-VALUES TO Item-1
        INITIALIZE Item-1 WITH FILLER
        CALL "COBDUMP" USING Item-1
        PERFORM 100-Init-Item-1
        DISPLAY " "

    DISPLAY "INITIALIZE Item-1 ALL TO VALUE"
        MOVE HIGH-VALUES TO Item-1
        INITIALIZE Item-1 ALPHANUMERIC TO VALUE
        CALL "COBDUMP" USING Item-1
```

```
            PERFORM 100-Init-Item-1
            DISPLAY " "

        DISPLAY "INITIALIZE Item-1 REPLACING NUMERIC BY 1"
            MOVE HIGH-VALUES TO Item-1
            INITIALIZE Item-1 REPLACING NUMERIC BY 1
            CALL "COBDUMP" USING Item-1
            PERFORM 100-Init-Item-1
            DISPLAY " "

        STOP RUN
        .

    100-Init-Item-1.
        MOVE HIGH-VALUES TO Item-1
        .
```

When executed, this program produces the following output:

```
MOVE HIGH-VALUES TO Item-1
<-Addr-> Byte <---------------- Hexadecimal ----------------> <---- Char ---->
======== ==== ================================================ ================
00404058    1 FF FF FF FF FF FF FF FF                        .........

INITIALIZE Item-1
<-Addr-> Byte <---------------- Hexadecimal ----------------> <---- Char ---->
======== ==== ================================================ ================
00404058    1 FF 30 00 20 20 2F 20 30 30                     .0.  / 00

INITIALIZE Item-1 WITH FILLER
<-Addr-> Byte <---------------- Hexadecimal ----------------> <---- Char ---->
======== ==== ================================================ ================
00404058    1 20 30 00 20 20 2F 20 30 30                      0.  / 00

INITIALIZE Item-1 ALL TO VALUE
<-Addr-> Byte <---------------- Hexadecimal ----------------> <---- Char ---->
======== ==== ================================================ ================
00404058    1 2A 2A FF 43 5A 5A 20 FF FF                     **.CZZ ..

INITIALIZE Item-1 REPLACING NUMERIC BY 1
<-Addr-> Byte <---------------- Hexadecimal ----------------> <---- Char ---->
======== ==== ================================================ ================
00404058    1 FF 31 01 FF FF FF FF 31 31                     .1.....11
```

7.8.24. INITIATE

INITIATE Syntax

```
INITIATE report-name-1
~~~~~~~~
```

The `INITIATE` statement starts Report-Writer Control System (RWCS) processing for a report.

1. Each *report-name-1* must be the name of a report having an `RD` (see [REPORT SECTION], page 135) defined for it.
2. The file in whose `FD` (see [File/Sort-Description], page 124) a `REPORT` *`report-name-1`* clause exists must be open for `OUTPUT` or `EXTEND` at the time the `INITIATE` statement is executed. See [OPEN], page 358, for more information on file open modes.
3. The `INITIATE` statement will initialize all of the following for each report named on the statement:
   - All sum counters, if any, will be set to 0
   - The report's `LINE-COUNTER` special register (see [Special Registers], page 265) will be set to 0
   - The report's `PAGE-COUNTER` special register will be set to 1
4. No report content will actually presented to the report file as a result of a successful `INITIATE` statement — that will not occur until the first `GENERATE` statement (see [GENERATE], page 332) is executed.

7.8.25. INSPECT

INSPECT Syntax

```
INSPECT { literal-1           }
~~~~~~~ { identifier-1        }
        { function-reference-1 }

 [ TALLYING { identifier-2 FOR { ALL|LEADING|TRAILING { literal-2    } }
   ~~~~~~~~                ~~~ { ~~~ ~~~~~~~ ~~~~~~~~ { identifier-3 } }
                               { CHARACTERS                           }
                                 ~~~~~~~~~~
             [ | { AFTER|BEFORE } INITIAL { literal-3    } | ] }...  ]
               |   ~~~~~ ~~~~~~           { identifier-4 } |

 [ REPLACING { { { ALL|FIRST|LEADING|TRAILING { literal-4    } }
   ~~~~~~~~~   { { ~~~ ~~~~~ ~~~~~~~ ~~~~~~~~ { identifier-5 } }
               { CHARACTERS                                   }
               { ~~~~~~~~~~                                   }

                 BY { [ ALL ] literal-5 }
                 ~~ {   ~~~             }
                    { identifier-6      }

             [ | { AFTER|BEFORE } INITIAL { literal-6    } | ] }...  ]
               |   ~~~~~ ~~~~~~           { identifier-7 } |

 [ CONVERTING { { literal-7    } TO { literal-8    }
   ~~~~~~~~~~   { identifier-8 } ~~ { identifier-9 }

             [ | { AFTER|BEFORE } INITIAL { literal-9     } | ] ]
               |   ~~~~~ ~~~~~~           { identifier-10 } |
```

The `INSPECT` statement is used to perform various counting and/or data-alteration operations against strings.

1. The reserved word `INITIAL` is optional and may be omitted. The presence or absence of this words has no effect upon the program.
2. If a `CONVERTING` clause is specified, neither the `TALLYING` nor `REPLACING` clauses may be used.
3. If either the `TALLYING` or `REPLACING` clauses are specified, the `CONVERTING` clause cannot be used.
4. If both the `TALLYING` and `REPLACING` clauses are specified, they must be specified in the order shown.

5. All literals and identifiers must be explicitly or implicitly defined as alphanumeric or alphabetic.
6. If *function-reference-1* is specified, it must be an invocation of an intrinsic function that returns a *string* result. Additionally, only the `TALLYING` clause may be specified.
7. If *literal-1* is specified, only the `TALLYING` clause may be specified.
8. Whichever is specified — *literal-1*, *identifier-1* or *function-reference-1* — that item will be referred to in the discussions that follows as the '*inspect subject*'.
9. The three optional clauses control the operation of this statement as follows:
   A. The `CONVERTING` clause replaces one or more individual characters found in the inspect subject with a different character in much the same manner as is possible with the `TRANSFORM` statement (see [TRANSFORM], page 411).
   B. The `REPLACING` clause replaces one or more sub strings located in the inspect subject with a different, but equally-sized replacement sub string. If you need to replace a sub string with another of a *different* length, consider using either the `SUBSTITUTE` intrinsic function (see [SUBSTITUTE], page 518) or the `SUBSTITUTE-CASE` intrinsic function (see [SUBSTITUTE-CASE], page 519).
   C. The `TALLYING` clause counts the number of occurrences of one or more strings of characters in the inspect subject.
10. The optional `INITIAL` clauses may be used to limit the range of characters in the inspect subject that the `CONVERTING`, `REPLACING` or `TALLYING` instruction in which they occur will apply. We call this the '*target range*' of the inspect subject. The target range is defined as follows:
    A. If there is no `INITIAL` clause specified, the target range is the entire inspect subject.
    B. Either a `BEFORE` phrase, an `AFTER` phrase or both may be specified. They may be specified in any order.
    C. The starting point of the target range will be the first character following the sub string identified by the `AFTER` specification. The ending point will be the last character immediately preceding the sub string identified by the `BEFORE` specification.
    D. If no `AFTER` is specified, the first character position of the target range will be character position #1 of the inspect subject.
    E. If no `BEFORE` is specified, the last character position of the target range will be the last character position of the inspect subject.
11. The following points apply to the use of the `TALLYING` clause:
    A. While there will typically be only be a single set of counting instructions on an `INSPECT`:

```
INSPECT Character-String
    TALLYING C-ABC FOR ALL "ABC"
```

    There could be multiple counting instructions specified:

```
INSPECT Character-String
    TALLYING C-ABC FOR ALL "ABC"
             C-BCDE FOR ALL "BCDE"
```

When there *are* multiple instructions, the one specified first will take priority over the one specified second, (and so forth) as the `INSPECT` proceeds forward through the inspect subject, character-by-character.

With the above example, if the inspect subject were `--ABCDEF----BCDEF--`, the final result of the counting would be that `C-ABC` would be incremented by 1 while `C-BCDE` would be incremented only once; although the human eye clearly sees two '`BCDE`' sequences, the `INSPECT ... TALLYING` would only "see" the second — the first would have been processed by the first (higher-priority) counting instruction.

B. Each set of counting instructions contains the following information:

   a. A target range, specified by the presence of an `AFTER INITIAL` and/or `BEFORE INITIAL` clause; the rules for specifying target ranges were covered previously.

   b. A Target Sub string — this is a sequence of characters to be located somewhere in the inspect subject and counted. Target sub strings may be defined as a literal value (figurative constants are allowed) or by the contents of an identifier. If the target sub string is specified as a figurative constant, it will be assumed to have a length of one ('`1`') character. The keywords before the literal or identifier control how many target sub strings could be identified from that replacement instruction, as follows:

      `ALL` — identifies every possible target sub string that occurs within the target range. There are three occurrences of `ALL 'XX'` found in `aXXabbXXccXXdd`.

      `LEADING` — identifies only an occurrence of the target sub string found either at the first character position of the target range or immediately following a previously-found occurrence. There are no occurrences of `LEADING 'XX'` found in `aXXabbXXccXXdd`, but there is one occurrence of `LEADING 'a'` (the first character).

      `TRAILING` — identifies only an occurrence of the target sub string found either at the very end of the target range or toward the end, followed by nothing but other occurrences. There are no occurrences of `LEADING 'XX'` found in `aXXabbXXccXXdd`, but there are two occurrences of `TRAILING 'd'`.

      The `CHARACTERS` option will match any one single character, regardless of what that character is.

C. *identifier-2* will be incremented by 1 each time the target sub string is found within the target range of the inspect subject. The `INSPECT` statement *will not* zero-out *identifier-2* at the start of execution of the `INSPECT` — it is the programmer's responsibility to ensure that all *identifier-2* data items are properly initialized to the desired starting values prior to execution of the `INSPECT`.

12. The following points apply to the use of the `REPLACING` clause:

A. While there will typically be only be a single set of replacement instructions on an `INSPECT`:

```
INSPECT Character-String
    REPLACING ALL "ABC" BY "DEF"
```

There could be multiple replacement instructions:

```
INSPECT Character-String
```

```
REPLACING ALL "ABC" BY "DEF"
          ALL "BCDE" BY "WXYZ"
```

When there *are* multiple replacement instructions, the one specified first will take priority over the one specified second, (and so forth) as the `INSPECT` proceeds forward through the inspect subject, character-by-character.

With the above example, if the inspect subject were `--ABCDEF----BCDEF--`, the final result of the replacement would be `--DEFDEF----WXYZF--`.

B. Each set of replacement instructions contains the following information:

   a. A target range, specified by the presence of an `AFTER INITIAL` and/or `BEFORE INITIAL` clause; the rules for specifying target ranges were covered previously.

   b. A Target Sub string — this is a sequence of characters to be located somewhere in the inspect subject and subsequently replaced with a new value. Target sub strings, which are specified before the `BY` keyword, may be defined as a literal value (figurative constants are allowed) or by the contents of an identifier. If the target sub string is specified as a figurative constant, it will be assumed to have a length of one ('`1`') character. The keywords before the literal or identifier control how many target sub strings could be identified from that replacement instruction, as follows:

      `ALL` — identifies every possible target sub string that occurs within the target range. There are three occurrences of `ALL 'XX'` found in `aXXabbXXccXXdd`.

      `FIRST` — the first occurrence of the target sub string found within the target range. The `FIRST 'XX'` found in `aXXabbXXccXXdd` would be the one found between the '`a`' and '`b`' characters.

      `LEADING` — an occurrence of the target sub string found either at the first character position of the target range or immediately following a previously-found occurrence. There are no occurrences of `LEADING 'XX'` found in `aXXabbXXccXXdd`, but there is one occurrence of `LEADING 'a'` (the first character).

      `TRAILING` — an occurrence of the target sub string found either at the very end of the target range or toward the end, followed by nothing but other occurrences. There are no occurrences of `LEADING 'XX'` found in `aXXabbXXccXXdd`, but there are two occurrences of `TRAILING 'd'`.

      The `CHARACTERS` option will match any one single character. When you use this option, the replacement sub string (see the next item) must be exactly one character in length.

   c. A Replacement Sub string — this is the sequence of characters that should replace the target sub string. Replacement sub strings are specified after the `BY` keyword. They too may be specified as a literal, either with or without an `ALL` prefix (again, figurative constants are allowed) or the value of an identifier. If a figurative constant is coded, the `ALL` keyword will be assumed, even if it wasn't specified. Literals without `ALL` will either be truncated or padded with spaces on the right to match the length of the target sub string. Literals with `ALL` or figurative constants will be repeated as necessary to match the length

of the target sub string. Identifiers specified as replacement sub strings must be defined with a length equal to that of the target sub string.

13. When both `REPLACING` and `TALLYING` are specified:
    A. The `INSPECT` statement will make a single pass through the sequence of characters comprising the inspect subject. As the pointer to the current inspect target character reaches a point where it falls within the explicit or implicit target ranges specified on the operational instructions of the two clauses, the actions specified by those instructions will become eligible to be taken. As the character pointer reaches a point where it falls past the end of target ranges, the instructions belonging to those target ranges will become disabled.
    B. At any point in time, there may well be multiple`REPLACING` and/or `TALLYING` operational instructions active. Only one of the `TALLYING` and one of the `REPLACING` instructions (if any) can be executed for any one character pointer position. In each case, it will be the first of the instructions in each category that produces a match with its target string specification.
    C. When both a `TALLYING` and a `REPLACING` instruction have been selected for execution, the `TALLYING` instruction will be executed first. This guarantees that `TALLYING` will compute occurrences based upon the *initial* value of the inspect subject before any replacements occur.
14. The following points apply to the use of the `CONVERTING` clause:
    A. A `CONVERTING` clause performs a series of single-character substitutions against a data item in much the same manner as is possible with the `TRANSFORM` statement (see [TRANSFORM], page 411).
    B. Unlike the `TALLYING` and `REPLACING` clauses, both of which may have multiple operations specified, there may be only one `CONVERTING` operation per `INSPECT`.
    C. If the length of *literal-7* or *identifier-8* (the "from" string) *exceeds* the length of *literal-8* or *identifier-9* (the "to" string), then the "to" string will be assumed to be padded to the right with enough spaces to make it the same length as the "from" string.
    D. If the length of the "from" string *is less than* the length of the "to" string, then the "to" string will be truncated to the length of the "from" string.
    E. Each character, in turn, within the "from" string will be searched for in the target range of the inspect subject. Each located occurrence will be replaced by the corresponding character of the "to" string.

7.8.26. MERGE

MERGE Syntax

```
MERGE sort-file-1
~~~~~
   { ON { ASCENDING  } KEY identifier-1...  }...
        { ~~~~~~~~~  }
        { DESCENDING }
          ~~~~~~~~~~
   [ WITH DUPLICATES IN ORDER ]
          ~~~~~~~~~~
   [ COLLATING SEQUENCE IS alphabet-name-1 ]
     ~~~~~~~~~
     USING file-name-1 file-name-2...
     ~~~~~
   { OUTPUT PROCEDURE IS procedure-name-1     }
   { ~~~~~~ ~~~~~~~~~                        }
   {       [ THRU|THROUGH procedure-name-2 ]  }
   {         ~~~~ ~~~~~~~                    }
   { GIVING file-name-3...                    }
   { ~~~~~~                                   }
```

The `DUPLICATES` clause is syntactically recognized but is otherwise non-functional.

The `MERGE` statement merges the contents of two or more files that have each been pre-sorted on a set of specified identical keys.

1. The reserved words `IN`, `IS`, `KEY`, `ON`, `ORDER`, `SEQUENCE` and `WITH` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved words `THRU` and `THROUGH` are interchangeable.
3. GnuCOBOL always behaves as if the `WITH DUPLICATES IN ORDER` clause is specified, even if it isn't.

   While any COBOL implementation's sort or merge facilities guarantee that records with duplicate key values will be in proper sequence with regard to other records with different key values, they generally make no promises as to the resulting relative sequence of records having duplicate key values with one another.

   Some COBOL implementations provide this optional clause to force their sort and merge facilities to retain duplicate key-value records in their original input sequence, relative to one another.
4. The *sort-file-1* named on the `MERGE` statement must be defined using a sort description (`SD` (see [File/Sort-Description], page 124)). This file is referred to in the remainder of this discussion as the *merge work file*.
5. Each *file-name-1*, *file-name-2* and *file-name-3* (if specified) must reference `ORGANIZATION LINE SEQUENTIAL` (see [ORGANIZATION LINE SEQUENTIAL],

page 111) or `ORGANIZATION SEQUENTIAL` (see [ORGANIZATION SEQUENTIAL], page 109) files. These files must be defined using a file description (`FD` (see [File/Sort-Description], page 124)).

6. The *identifier-1* ... field(s) must be defined as field(s) within a record of *sort-file-1*.
7. The record descriptions of *file-name-1*, *file-name-2*, *file-name-3* (if any) and *sort-file-1* are assumed to be identical in layout and size. While the actual data names used for fields in these files' records may differ, the structure of records, `PICTURE` (see [PICTURE], page 194) of fields, `USAGE` (see [USAGE], page 232) of fields, size of fields and location of fields within the records should match field-by-field across all files, at least as far as the `KEY` fields are concerned.
8. A common programming technique when using the `MERGE` statement is to define the records of all files involved as simple elementary items of the form `01 record-name PIC X(n).` where n is the record size. The only file where records are actually described in detail would then be *sort-file-1*.
9. The following rules apply to the files named on the `USING` clause:
   A. None of them may be open at the time the `MERGE` is executed.
   B. Each of those files is assumed to be already sorted according to the specifications set forth on the `MERGE` statement's `KEY` clause.
   C. No two of those files may be referenced on a `SAME RECORD AREA` (see [SAME RECORD AREA], page 117), `SAME SORT AREA` or `SAME SORT-MERGE AREA` statement.
10. The merging process is as follows:
    A. As the `MERGE` statement begins execution, the first record in each of the `USING` files is read automatically.
    B. As the `MERGE` statement executes, the current record from each of the `USING` files is examined and compared to each other according to the rules set forth by the `KEY` clause and the alphabet (see [Alphabet-Name-Clause], page 96) specified on the `COLLATING SEQUENCE` clause. The record that should be next in sequence will be written to the merge work file and the `USING` file from which that record came will be read so that its next record is available. As end-of-file conditions are reached on `USING` files, those files will be excluded from further processing — processing continues with the remaining files until all the contents of all of them have been exhausted.
    C. After the merge work file has been populated, the merged data will be written to each *file-name-3* if the `GIVING` clause was specified, or will be processed by utilizing an `OUTPUT PROCEDURE`.
    D. When `GIVING` is specified, none of the *file-name-3* files can be open at the time the `MERGE` statement is executed.
    E. When an output procedure is used, the procedure(s) specified on the `OUTPUT PROCEDURE` clause will be invoked as if by a procedural `PERFORM` (see [Procedural PERFORM], page 360) statement with no `VARYING`, `TIMES` or `UNTIL` options specified. Merged records may be read from the merge work file — one at a time — within the output procedure using the `RETURN` (see [RETURN], page 373) statement.

A `GO TO` statement (see [GO TO], page 335) that transfers control out of the output procedure will terminate the `MERGE` statement but allows the program to continue executing from the point where the `GO TO` statement transferred control to. Once an output procedure has been "aborted" using a `GO TO` it cannot be resumed, and the contents of the merge work file are lost. You may, however, re-execute the `MERGE` statement itself. *Using a* `GO TO` statement *to prematurely terminate a merge, or re-starting a previously-cancelled merge is not considered good programming style and should be avoided.*

An output procedure should be terminated in the same way a procedural `PERFORM` statement would be. Usually, this action will be taken once the `RETURN` statement indicates that all records in the merge work file have been processed, but termination could occur at *any* time — via an `EXIT` statement (see [EXIT], page 328) — if required.

Neither a file-based `SORT` statement (see [File-Based SORT], page 391) nor another `MERGE` statement may be executed within the scope of the procedures comprising the output procedure unless those statements utilize a different sort or merge work file.

F. Once the output procedure terminates, or the last *file-name-3* file has been populated with merged data, the output phase — and the `MERGE` statement itself — is complete.

7.8.27. MOVE

7.8.27.1. Simple MOVE

Simple MOVE Syntax

```
MOVE { literal-1    } TO identifier-2...
~~~~ { identifier-1 } ~~
```

The Simple `MOVE` statement moves a specific value to one or more receiving data items.

1. The `MOVE` statement will replace the contents of one or more receiving data items (*identifier-2*) with a new value — the one specified by *literal-1* or *identifier-1*.
2. Only numeric data can be moved to a numeric or numeric-edited *identifier-2*. A `MOVE` involving numeric data will perform any necessary format conversions that might be necessary due to differing `USAGE` (see [USAGE], page 232) specifications.
3. The contents of the *identifier-1* data item will not be changed, unless that same data item appears as an *identifier-2*. Note that such situations will cause a warning message to be issued by the compiler, if warning messages are enabled.

7.8.27.2. MOVE CORRESPONDING

MOVE CORRESPONDING Syntax

```
MOVE CORRESPONDING identifier-1 TO identifier-2...
~~~~ ~~~~                       ~~
```

The `MOVE CORRESPONDING` statement similarly-named items from one group item to another.

1. The reserved word `CORRESPONDING` may be abbreviated as `CORR`.
2. Both *identifier-1* and *identifier-2* must be group items.
3. See [CORRESPONDING], page 258, for a discussion of how corresponding matches between two group items are established.
4. When corresponding matches are established, the effect of a `MOVE CORRESPONDING` on those matches will be as if a series of individual `MOVE`s were done — one for each match.

7.8.28. MULTIPLY

7.8.28.1. MULTIPLY BY

MULTIPLY BY Syntax

```
MULTIPLY { literal-1    } BY { identifier-2
~~~~~~~~ { identifier-1 } ~~

     [ ROUNDED [ MODE IS { AWAY-FROM-ZERO         } ] ] }...
       ~~~~~~~   ~~~~    { ~~~~~~~~~~~~~~         }
                         { NEAREST-AWAY-FROM-ZERO }
                         { ~~~~~~~~~~~~~~~~~~~~~~ }
                         { NEAREST-EVEN           }
                         { ~~~~~~~~~~~~           }
                         { NEAREST-TOWARD-ZERO    }
                         { ~~~~~~~~~~~~~~~~~~     }
                         { PROHIBITED             }
                         { ~~~~~~~~~~             }
                         { TOWARD-GREATER         }
                         { ~~~~~~~~~~~~~~         }
                         { TOWARD-LESSER          }
                         { ~~~~~~~~~~~~~          }
                         { TRUNCATION             }
                           ~~~~~~~~~~

   [ ON SIZE ERROR imperative-statement-1 ]
        ~~~~ ~~~~~

   [ NOT ON SIZE ERROR imperative-statement-2 ]
     ~~~    ~~~~ ~~~~~

[ END-MULTIPLY ]
  ~~~~~~~~~~~~
```

The `MULTIPLY BY` statement computes the product of one or more data items (*identifier-2*) and either a numeric literal or another data item.

1. The reserved words `IS` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. Both *identifier-1* and *identifier-2* must be numeric un-edited data items; *literal-1* must be a numeric literal.
3. The product of *identifier-1* or *literal-1* and each *identifier-2*, in turn, will be computed and moved to each of the *identifier-2* data items, replacing the prior contents.
4. The value of *identifier-1* is not altered, unless that same data item appears as an *identifier-2*.
5. The optional `ROUNDED` (see [ROUNDED], page 261) clause available to each *identifier-2*

will control how non-integer results will be saved.

6. The optional `ON SIZE ERROR` and `NOT ON SIZE ERROR` clauses may be used to detect and react to the failure or success, respectively, of an attempt to perform a calculation. In this case, failure is defined as being an *identifier-2* with an insufficient number of digit positions available to the left of any implied decimal point. See [ON SIZE ERROR + NOT ON SIZE ERROR], page 261, for additional information.

7.8.28.2. MULTIPLY GIVING

MULTIPLY GIVING Syntax

```
MULTIPLY { literal-1    } BY { literal-2    } GIVING { identifier-3
~~~~~~~~ { identifier-1 } ~~ { identifier-2 } ~~~~~~

     [ ROUNDED [ MODE IS { AWAY-FROM-ZERO         } ] ] }...
       ~~~~~~~   ~~~~    { ~~~~~~~~~~~~~~         }
                         { NEAREST-AWAY-FROM-ZERO }
                         { ~~~~~~~~~~~~~~~~~~~~~~ }
                         { NEAREST-EVEN           }
                         { ~~~~~~~~~~~~           }
                         { NEAREST-TOWARD-ZERO    }
                         { ~~~~~~~~~~~~~~~~~~     }
                         { PROHIBITED             }
                         { ~~~~~~~~~~             }
                         { TOWARD-GREATER         }
                         { ~~~~~~~~~~~~~~         }
                         { TOWARD-LESSER          }
                         { ~~~~~~~~~~~~~          }
                         { TRUNCATION             }
                           ~~~~~~~~~~

   [ ON SIZE ERROR imperative-statement-1 ]
        ~~~~ ~~~~~

   [ NOT ON SIZE ERROR imperative-statement-2 ]
     ~~~    ~~~~ ~~~~~

[ END-MULTIPLY ]
  ~~~~~~~~~~~~
```

The `MULTIPLY GIVING` statement computes the product of two literals and/or data items and saves that result in one or more other data items.

1. The reserved words `IS` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. Both *identifier-1* and *identifier-2* must be numeric un-edited data items; *literal-1* and *literal-2* must be numeric literals.
3. The product of *identifier-1* or *literal-1* and *identifier-2* or *literal-2* will be computed and moved to each of the *identifier-3* data items, replacing their old contents.
4. Neither the value of *identifier-1* nor *identifier-2* will be altered, unless either appears as an *identifier-3*.
5. The optional `ROUNDED` (see [ROUNDED], page 261) clause available to each *identifier-2* will control how non-integer results will be saved.

6. The optional `ON SIZE ERROR` and `NOT ON SIZE ERROR` clauses may be used to detect and react to the failure or success, respectively, of an attempt to perform a calculation. In this case, failure is defined as being an *identifier-2* with an insufficient number of digit positions available to the left of any implied decimal point. See [ON SIZE ERROR + NOT ON SIZE ERROR], page 261, for additional information.

7.8.29. OPEN

OPEN Syntax

```
OPEN { { INPUT  } [ SHARING WITH { ALL OTHER } ] file-name-1
~~~~   { ~~~~~  }   ~~~~~~~      { ~~~       }
       { OUTPUT }                { NO OTHER  }
       { ~~~~~~ }                { ~~        }
       { I-O    }                { READ ONLY }
       { ~~~    }                  ~~~~ ~~~~
       { EXTEND }
         ~~~~~~
     [ { REVERSED           } ] }...
       { ~~~~~~~~           }
       { WITH { NO REWIND } } }
       {      { ~~ ~~~~~~ } } }
       {      { LOCK      } } }
                ~~~~
```

The `NO REWIND`, and `REVERSED` clauses are syntactically recognized but are otherwise non-functional.

The `OPEN` statement makes one or more files described in your program available for use.

1. The reserved words `OTHER` and `WITH` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The `SHARING` and `WITH LOCK` clauses may not both be specified in the same `OPEN` statement.
3. Any file defined in a GnuCOBOL program must be successfully opened before it or any of its record descriptions may be referenced on:

   A `CLOSE` statement (see [CLOSE], page 299)

   A `DELETE` statement (see [DELETE], page 304)

   A `READ` statement (see [READ], page 366)

   A `REWRITE` statement (see [REWRITE], page 374)

   A `START` statement (see [START], page 397)

   An `UNLOCK` statement (see [UNLOCK], page 412)

   A `WRITE` statement (see [WRITE], page 417)
4. Any attempt to open a file that is already open will fail with a file status of 41 (see [File Status Codes], page 107).
5. Any open failure (including status 41) may be trapped using `DECLARATIVES` (see [DECLARATIVES], page 254) or an error procedure established using the `CBL_ERROR_PROC` built-in system subroutine (see [CBL_ERROR_PROC], page 568) built-in subroutine or even just checking the status field defined. It is up to the

programmer to check for bad statuses and respond accordingly such as issue a CLOSE before dealing with the problem.

6. The `INPUT`, `OUTPUT`, `I-O` and `EXTEND` open modes inform GnuCOBOL of the manner in which you wish to use the file, as follows:

   `INPUT` You may only read the existing contents of the file — only the `CLOSE`, `READ`, `START` and `UNLOCK` statements will be allowed. This enforcement takes place at execution time, not compilation time.

   `OUTPUT` You may only write new content (which will completely replace any previous file contents) to the file — only the `CLOSE`, `UNLOCK` and `WRITE` statements will be allowed. This enforcement takes place at execution time, not compilation time.

   `I-O` You may perform any operation you wish against the file — all file I/O statements will be allowed.

   `EXTEND` You may only write new content (which will be appended after the previously existing file contents) to the file — only the `CLOSE`, `UNLOCK` and `WRITE` statements will be allowed. This enforcement takes place at execution time, not compilation time. You cannot extend an empty file; this will not generate a runtime error, but no output will appear in the file.

7. The `SHARING` clause informs the GnuCOBOL file runtime modules how you are willing to co-exist with any other GnuCOBOL programs that may attempt to open the same file after your program does. See [File Sharing], page 57, for an explanation of the `SHARING` clause.

8. The `WITH LOCK` option will be functional only if your GnuCOBOL build can support it. GnuCOBOL built for MinGW or native Windows will not, because the Unix `fcntl` primitive doesn't exist in those environments. GnuCOBOL built for Cygwin or Unix will.

7.8.30. PERFORM

7.8.30.1. Procedural PERFORM

Procedural PERFORM Syntax

```
PERFORM procedure-name-1 [ THRU|THROUGH procedure-name-2 ]
~~~~~~~                  ~~~~ ~~~~~~~
   [ { [ WITH TEST { BEFORE } ] { VARYING-Clause                } } ]
     {        ~~~~ { ~~~~~~ }   { UNTIL conditional-expression-1 } }
     {             { AFTER  }     ~~~~~                           }
     {               ~~~~~                                       }
     { UNTIL EXIT|FOREVER                                        }
     { ~~~~~ ~~~~ ~~~~~~~                                        }
     { { literal-1    } TIMES                                    }
     { { identifier-1 } ~~~~~                                    }
```

This format of the `PERFORM` statement is used to transfer control to one or more procedures, which will return control back when complete. Execution of the procedure(s) can be done a single time, multiple times, repeatedly until a condition becomes `TRUE` or forever (with some way of breaking out of the control of the `PERFORM` or of halting program execution within the procedure(s)).

1. The reserved word `WITH` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. The reserved words `THRU` and `THROUGH` are interchangeable.
3. The reserved word and phrase `FOREVER` and `UNTIL EXIT` are interchangeable.
4. Both *procedure-name-1* and *procedure-name-2* must be procedure division sections or paragraphs defined in the same program as the `PERFORM` statement. If *procedure-name-2* is specified, it must follow *procedure-name-1* in the program's source code.
5. The *perform scope* is defined as being the statements within *procedure-name-1*, the statements within *procedure-name-2* and all statements in all procedures defined between them.
6. *literal-1* must be a numeric literal or a reference to a function that returns a numeric value. The value must be an integer greater than zero.
7. *identifier-1* must be an elementary un-edited numeric data item with an integer value greater than zero.
8. Without the `UNTIL`, `UNTIL EXIT`, `TIMES`, *VARYING-Clause* (see [VARYING], page 363) or `FOREVER` clauses, the code within the perform scope will be executed once, after which control will return to the statement following the `PERFORM`.
9. The `FOREVER` option will repeatedly execute the code within the perform scope with no conditions defined for termination of the repetition — it will be up to the program-

mer to include an `EXIT SECTION` statement (see [EXIT], page 328) or `EXIT PARAGRAPH` statement within the procedure(s) being performed that will break out of the loop.

10. The `TIMES` option will repeat the execution of the code within the perform scope a fixed number of times. When the `PERFORM` statement begins execution, an internal repeat counter (not accessible to the programmer) will be set to the value of *literal-1* or the value within *identifier-1*.

    If the counter has a value greater than zero, the statement(s) within the `PERFORM` scope will be executed, after which the counter will be decremented by 1 with each repetition. Once that counter reaches zero, repetition will cease and control will fall into the next statement following the `PERFORM`.

    If the *identifier-1* option was used, altering the value of that data item within the perform scope will *not* affect the repetition count.

11. The `UNTIL` *`conditional-expression-1`* option will repeat the code within the perform scope until the specified conditional expression evaluates to a `TRUE` value.

12. The optional `WITH TEST` clause will control whether `UNTIL` testing occurs `BEFORE` the statements within the perform scope are executed on each iteration (creating the possibility — if *conditional-expression-1* is initially `TRUE` — that the statements within the perform scope will never be executed) or `AFTER` (guaranteeing the statements within the perform scope will be executed at least once).

    The default, if this clause is absent, is `WITH TEST BEFORE`.

    This clause may not be coded when the `TIMES` clause is used.

13. The optional *`VARYING-Clause`* is a mechanism that creates an advanced loop-management mechanism complete with one or more numeric data items being automatically incremented (or decremented) on each loop iteration as well as the termination control of an `UNTIL` clause. See [VARYING], page 363, for the details.

7.8.30.2. Inline PERFORM

Inline PERFORM Syntax

```
PERFORM
~~~~~~~
 [ { [ WITH TEST { BEFORE } ] { VARYING-Clause                 } } ]
   {        ~~~~ { ~~~~~~ }   { UNTIL conditional-expression-1 } }
   {             { AFTER  }     ~~~~~                           }
   {               ~~~~~                                       }
   { UNTIL EXIT|FOREVER                                        }
   { ~~~~~ ~~~~ ~~~~~~~                                        }
   { { literal-1    } TIMES                                    }
   { { identifier-1 } ~~~~~                                    }

   imperative-statement-1

[ END-PERFORM ]
  ~~~~~~~~~~~
```

This format of the `PERFORM` statement is identical in operation to the procedural `PERFORM`, except for the fact that the statement(s) comprising the perform scope (*imperative-statement-1*) (see [Imperative Statement], page 722) are now specified in-line with the `PERFORM` code rather than in procedures located elsewhere within the program.

7.8.30.3. VARYING

VARYING Syntax

```
VARYING identifier-2 FROM { literal-2    } [ BY { literal-3    } ]
~~~~~~~              ~~~~ { identifier-3 }   ~~ { identifier-4 }
        [ UNTIL conditional-expression-1 ]
          ~~~~~
[ AFTER identifier-5 FROM { literal-4    } [ BY { literal-5    } ]
  ~~~~~              ~~~~ { identifier-6 }   ~~ { identifier-7 }
        [ UNTIL conditional-expression-2 ] ]...
          ~~~~~
```

The `VARYING` clause, available on both formats of the `PERFORM` statement, is a looping mechanism that allows for the specification of one or more numeric data items that will be initialized to a programmer-specified value and automatically incremented by another programmer-specified value after each loop iteration.

1. All identifiers used in a *VARYING-Clause* must be elementary, un-edited numeric data items. All literals must be numeric literals.
2. The following points describe the sequence of events that take place as a result of the `VARYING` portion of the clause:
   A. When the `PERFORM` begins execution, the `FROM` value will be moved to *identifier*.
   B. If the `PERFORM` specifies or implies `WITH TEST BEFORE`, *conditional-expression-1* will be evaluated and processing of the `PERFORM` will halt if the expression evaluates to `TRUE`. If `WITH TEST BEFORE` was *not* specified or implied, or if the conditional expression evaluated to `FALSE`, processing proceeds with step C.
   C. The statements within the perform scope will be executed. If a `GO TO` executed within the perform scope transfers control to a point outside the perform scope, processing of the `PERFORM` will halt.
   D. When the statements within the perform scope terminate the loop iteration, by one of:
      - allowing the flow of execution to attempt to fall past the last statement in the perform scope
      - executing an `EXIT PERFORM CYCLE` statement (see [EXIT], page 328)
      - executing an `EXIT PARAGRAPH` statement or `EXIT SECTION` statement when there is only one paragraph (or section) in the perform scope (this option only applies to a procedural `PERFORM`)

      If `WITH TEST AFTER` was specified, control will return back to the `PERFORM`, where *conditional-expression-1* will be evaluated, and processing of the `PERFORM` will halt if the expression evaluates to `TRUE`. If `WITH TEST AFTER` was *not* specified, or if the conditional expression evaluated to `FALSE`, processing continues with the next step.

E. The BY value, if any, will be added to *identifier-2*. If no BY is specified, it will be treated as if BY 1 had been specified.

F. Return to step C.

3. Most *VARYING-Clause*s have no AFTER specified. Those that do, however, are establishing a loop-within-a-loop situation where the process described above in steps ('A') through ('F') will take place from the AFTER, and those six processing steps actually replace step C of the VARYING. This "nesting" process can continue indefinitely, with each additional AFTER.

An example should really help you see this at work. Observe the following code which defines a two-dimensional (3 row by 4 column) table and a pair of numeric data items to be used to subscript references to each element of the table:

```
01  PERFORM-DEMO.
    05 PD-ROW              OCCURS 3 TIMES.
       10 PD-COL           OCCURS 4 TIMES
          15 PD            PIC X(1).
01  PD-Col-No              PIC 9 COMP.
01  PD-Row-No              PIC 9 COMP.
```

Let's say the 3x4 "grid" defined by the above structure has these values:

```
A B C D
E F G H
I J K L
```

This code will display ABCDEFGHIJKL on the console output window:

```
PERFORM WITH TEST AFTER
        VARYING PD-Row-No FROM 1 BY 1 UNTIL PD-Row-No = 3
          AFTER PD-Col-No FROM 1 BY 1 UNTIL PD-Col-No = 4
    DISPLAY PD (PD-Row-No, PD-Col-No) WITH NO ADVANCING
END-PERFORM
```

While this code will display AEIBFJCGKDHL on the console output window:

```
PERFORM WITH TEST AFTER
        VARYING PD-Col-No FROM 1 BY 1 UNTIL PD-Col-No = 4
          AFTER PD-Row-No FROM 1 BY 1 UNTIL PD-Row-No = 3
    DISPLAY PD (PD-Row-No, PD-Col-No) WITH NO ADVANCING
END-PERFORM
```

While we're looking at sample code, this code displays ABCEFG:

```
PERFORM
        VARYING PD-Row-No FROM 1 BY 1 UNTIL PD-Row-No = 3
          AFTER PD-Col-No FROM 1 BY 1 UNTIL PD-Col-No = 4
    DISPLAY PD (PD-Row-No, PD-Col-No) WITH NO ADVANCING
END-PERFORM
```

By removing the WITH TEST clause, the statement is now assuming WITH TEST BEFORE. Since testing now happens *before* the DISPLAY statement gets executed, when PD-Row-No is 3 and PD-Col-No is 4 the DISPLAY statement won't be executed.

Most COBOL programmers, when using `WITH TEST BEFORE` explicitly or implicitly have developed the habit of using '`>`' rather than '`=`' on `UNTIL` clauses. This would make the sample code:

```
PERFORM
        VARYING PD-Row-No FROM 1 BY 1 UNTIL PD-Row-No > 3
          AFTER PD-Col-No FROM 1 BY 1 UNTIL PD-Col-No > 4
    DISPLAY PD (PD-Row-No, PD-Col-No) WITH NO ADVANCING
END-PERFORM
```

With this change, `ABCDEFGHIJKL` is once again displayed.

7.8.31. READ

7.8.31.1. Sequential READ

Sequential READ Syntax

```
READ file-name-1 [ { NEXT|PREVIOUS } ] RECORD [ INTO identifier-1 ]
~~~~               { ~~~~ ~~~~~~~~ }           ~~~~
 [ { IGNORING LOCK    } ]
   { ~~~~~~~~ ~~~~    }
   { WITH [ NO ] LOCK }
   {        ~~   ~~~~ }
   { WITH KEPT LOCK   }
   {      ~~~~ ~~~~   }
   { WITH IGNORE LOCK }
   {      ~~~~~~ ~~~~ }
   { WITH WAIT        }
          ~~~~
 [ AT END imperative-statement-1 ]
      ~~~
 [ NOT AT END imperative-statement-2 ]
   ~~~    ~~~
[ END-READ ]
  ~~~~~~~~
```

This form of the `READ` statement retrieves the next (or previous) record from a file.

1. The reserved words `AT`, `RECORD` and `WITH` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The *file-name-1* file *must* have been defined via an `FD` (see [File/Sort-Description], page 124), not an `SD`.
3. The *file-name-1* file must currently be open for `INPUT` (see [File OPEN Modes], page 359) or `I-O`.
4. If *file-name-1* is an `ORGANIZATION RELATIVE` (see [ORGANIZATION RELATIVE], page 113) or `ORGANIZATION INDEXED` (see [ORGANIZATION INDEXED], page 115) file with an `ACCESS MODE RANDOM`, this statement cannot be used.
5. If *file-name-1* was specified as `ACCESS MODE SEQUENTIAL`, this is the *only* format of the `READ` statement that is available.
6. If *file-name-1* is an `ORGANIZATION RELATIVE` (see [ORGANIZATION RELATIVE], page 113) or `ORGANIZATION INDEXED` (see [ORGANIZATION INDEXED], page 115) file with `ACCESS MODE DYNAMIC`, this statement as well as a random `READ` (see [Random READ], page 368) may be used.
7. The keywords `NEXT` and `PREVIOUS` specify what "direction of travel" the reading process will take through the file. If neither is specified, `NEXT` is assumed.

8. The `PREVIOUS` option is available only for `ORGANIZATION INDEXED` files.
9. When reading any sequential (any organization) or relative file, the "next" direction refers to the physical sequence of records in the file. When reading an indexed file, the "next" and "previous" directions refer to the sequence of primary or alternate record key values in the file's records, regardless of where the records physically occur within the file.
10. The minimal statement `READ` *`file-name-1`* is perfectly legal according to *both* READ formats. For that reason, when `ACCESS MODE DYNAMIC` has been specified and you want to tell the GnuCOBOL compiler that this minimal statement should be treated as a *sequential* `READ`, you must add either `NEXT` or `PREVIOUS` to the statement (otherwise it will be treated as a random `READ`).
11. A successful sequential READ will retrieve the next available record from *file-name-1*, in either a "next" or "previous" direction from the most-recently-read record, depending upon the use of the `NEXT` or `PREVIOUS` option. The newly-retrieved record data will be saved into the 01-level record structure(s) that immediately follow the file's `FD`. If the optional `INTO` clause is present, a copy of the just-retrieved record will be automatically moved to *identifier-1*.
12. When an `ORGANIZATION RELATIVE` file has been successfully read, the file's `RELATIVE KEY` (see [ORGANIZATION RELATIVE], page 113) field will be automatically populated with the relative record number (ordinal occurrence number) of the record in the file.
13. The optional `LOCK` options may be used to manually control access to the retrieved record by other programs while this program is running. See [Record Locking], page 59, to review the various record locking behaviours.
14. The optional `AT END` clause, if coded, is used to detect and react to the failure of an attempt to retrieve another record from the file due to an end-of-file (i.e. no more records) condition.
15. The optional `NOT AT END` clause, if coded, will check for a file status value of 00. See [File Status Codes], page 107, for additional information.

7.8.31.2. Random READ

Random READ Syntax

```
READ file-name-1 RECORD [ INTO identifier-1 ]
~~~~                     ~~~~

 [ { IGNORING LOCK    } ]
   { ~~~~~~~~ ~~~~    }
   { WITH [ NO ] LOCK }
   {        ~~   ~~~~ }
   { WITH KEPT LOCK   }
   {      ~~~~ ~~~~   }
   { WITH IGNORE LOCK }
   {      ~~~~~~ ~~~~ }
   { WITH WAIT        }
          ~~~~

 [ KEY IS identifier-2 ]
   ~~~

 [ INVALID KEY imperative-statement-1 ]
   ~~~~~~~

 [ NOT INVALID KEY imperative-statement-2 ]
   ~~~ ~~~~~~~

[ END-READ ]
  ~~~~~~~~
```

This form of the `READ` statement retrieves an arbitrary record from an `ORGANIZATION RELATIVE` (see [ORGANIZATION RELATIVE], page 113) or `ORGANIZATION INDEXED` (see [ORGANIZATION INDEXED], page 115) file.

1. The reserved words `IS`, `KEY` (on the `INVALID` and `NOT INVALID` clauses), `RECORD` and `WITH` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The *file-name-1* file *must* have been defined via an `FD` (see [File/Sort-Description], page 124), not an `SD`.
3. The *file-name-1* file must currently be open for `INPUT` (see [File OPEN Modes], page 359) or `I-O`.
4. If the `ACCESS MODE` of *file-name-1* is `SEQUENTIAL`, or the `ORGANIZATION` of the file is any form of sequential, this format of the `READ` statement cannot be used.
5. If the `ACCESS MODE` of *file-name-1* is `RANDOM`, this is the *only* format of the `READ` statement that is available.
6. If *file-name-1* is an `ORGANIZATION RELATIVE` (see [ORGANIZATION RELATIVE], page 113) or `ORGANIZATION INDEXED` (see [ORGANIZATION INDEXED], page 115) file with `ACCESS MODE DYNAMIC`, this statement as well as a sequential `READ` (see [Sequential READ], page 366) may be used.

7. The minimal statement `READ file-name-1` is perfectly legal according to *both* READ formats. For that reason, when `ACCESS MODE DYNAMIC` has been specified and you want to tell the GnuCOBOL compiler that this minimal statement should be treated as a *random* `READ`, you must omit the `NEXT` or `PREVIOUS` available to the sequential format of the `READ` statement to ensure the statement *will* be treated as a random `READ`.
8. The optional `KEY` clause tells the compiler how a record is to be located in the file.

   If the `KEY` clause is absent, and the file is

   `ORGANIZATION RELATIVE`
   : the contents of the field declared as the file's `RELATIVE KEY` will be used to identify a record

   `ORGANIZATION INDEXED`
   : the contents of the field declared as the file's `RECORD KEY` will be used to identify a record.

   If the `KEY` clause *is* specified, and the file is

   `ORGANIZATION RELATIVE`
   : the contents of *identifier-2* will be used as the relative record number of the record to be accessed. *identifier-2* need not be the `RELATIVE KEY` (see [ORGANIZATION RELATIVE], page 113) field of the file (although it could be if you wish).

   `ORGANIZATION INDEXED`
   : *identifier-2 must* be the `RECORD KEY` (see [ORGANIZATION INDEXED], page 115) or one of the file's `ALTERNATE RECORD KEY` fields (if any). The current contents of that field will identify the record to be accessed. If an alternate record key is used, and that key allows duplicate values, the record accessed will be the *first* one having that key value.
9. Once read from the file, the newly-retrieved record data will be saved into the 01-level record structure(s) that immediately follow the file's `FD`. If the optional `INTO` clause is present, a copy of the just-retrieved record will be automatically moved to *identifier-1*.
10. When an `ORGANIZATION RELATIVE` file has been successfully read, the file's `RELATIVE KEY` (see [ORGANIZATION RELATIVE], page 113) field will be automatically populated with the relative record number (ordinal occurrence number) of the record in the file.
11. The optional `LOCK` options may be used to manually control access to the retrieved record by other programs while this program is running. See [Record Locking], page 59, to review the various record locking behaviours.
12. The optional `INVALID KEY` and `NOT INVALID KEY` clauses may be used to detect and react to the failure or success, respectively, by detecting non-zero (typically 23 = key not found = record not found) and 00 file status codes, respectively. See [File Status Codes], page 107, for additional information.

7.8.32. READY TRACE

READY TRACE Syntax

```
READY TRACE
~~~~~ ~~~~~
```

The `READY TRACE` statement turns procedure or procedure-and-statement tracing on.

1. In order for this statement to be functional, tracing code must have been generated into the compiled program using either the `-ftrace` switch (procedures only) or `-ftraceall` switch (procedures and statements).
2. Tracing may be turned off at any point by executing the `RESET TRACE` statement (see [RESET TRACE], page 372).
3. The `COB_SET_TRACE` run-time environment variable (see [Run Time Environment Variables], page 658) provides another way to control tracing. If this environment variable is set to a value of '`Y`' prior to the start of program execution, tracing starts at the point the program begins execution, as if `READY TRACE` were the first executed statement.

7.8.33. RELEASE

RELEASE Syntax

```
RELEASE record-name-1 [ FROM { literal-1    } ]
~~~~~~~                 ~~~~ { identifier-1 }
```

The `RELEASE` statement adds a new record to a sort work file.

1. This statement is valid only within the `INPUT PROCEDURE` of a file-based `SORT` statement (see [File-Based SORT], page 391).
2. The specified *record-name-1* must be a record defined to the sort description (`SD` (see [File/Sort-Description], page 124)) of the sort work file being processed by the current sort.
3. The optional `FROM` clause will cause *literal-1* or *identifier-1* to be automatically moved into *record-name-1* prior to writing *record-name-1*'s contents to the *file-name-1*. If this clause is not specified, it is the programmer's responsibility to populate *record-name-1* with the desired data prior to executing the `RELEASE`.

7.8.34. RESET TRACE

RESET TRACE Syntax

```
RESET TRACE
~~~~~ ~~~~~
```

The `RESET TRACE` statement turns procedure or procedure-and-statement tracing off.

1. By default, procedure and procedure-and-statement tracing is off as programs begin execution. The `READY TRACE` statement (see [READY TRACE], page 370) can be used to turn tracing on.
2. In order for this statement to be functional, tracing code must have been generated into the compiled program using either the `-ftrace` switch (procedures only) or `-ftraceall` switch (procedures and statements).
3. The `COB_SET_TRACE` run-time environment variable (see [Run Time Environment Variables], page 658) provides another way to control tracing. If this environment variable is set to a value of '`Y`' prior to the start of program execution, tracing started at the point the program begins execution, as if `READY TRACE` were the first executed statement. The `RESET TRACE` statement, if executed, will then turn off tracing.

7.8.35. RETURN

RETURN Syntax

```
RETURN sort-file-name-1 RECORD
~~~~~~
 [ INTO identifier-1 ]
   ~~~~
   AT END imperative-statement-1
      ~~~
 [ NOT AT END imperative-statement-2 ]
   ~~~    ~~~
[ END-RETURN ]
  ~~~~~~~~~~
```

The `RETURN` statement reads a record from a sort or merge work file.

1. The reserved words `AT` and `RECORD` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The `RETURN` statement is valid only within the `OUTPUT PROCEDURE` of a file-based `SORT` (see [File-Based SORT], page 391) or a `MERGE` statement (see [MERGE], page 349) statement.
3. The *sort-file-name-1* file must be a sort- or merge work file defined with a `SD` (see [File/Sort-Description], page 124), not an `FD`.
4. A successful `RETURN` will retrieve the next available record from *sort-file-name-1*. The newly-retrieved record data will be saved into the 01-level record structure(s) that immediately follow the file's SD. If the optional `INTO` clause is present, a copy of the just-retrieved record will be automatically moved to *identifier-1*.
5. The mandatory `AT END` clause is used to detect and react to the failure of an attempt to retrieve another record from the file due to an end-of-file (i.e. no more records) condition.
6. The optional `NOT AT END` clause, if coded, will check checking for a file status value of 00. See [File Status Codes], page 107, for additional information.

7.8.36. REWRITE

REWRITE Syntax

```
REWRITE record-name-1
~~~~~~~
   [ FROM { literal-1    } ]
     ~~~~ { identifier-1 }

   [ WITH [ NO ] LOCK ]
            ~~   ~~~~

   [ INVALID KEY imperative-statement-1 ]
     ~~~~~~~

   [ NOT INVALID KEY imperative-statement-2 ]
     ~~~ ~~~~~~~

[ END-REWRITE ]
  ~~~~~~~~~~
```

The `REWRITE` statement replaces a logical record on a disk file.

1. The reserved words `KEY` and `WITH` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The *record-name-1* specified on the statement must be defined as an 01-level record subordinate to the File Description (`FD` (see [File/Sort-Description], page 124)) of a file that is currently open for `I-O` (see [File OPEN Modes], page 359).
3. The optional `FROM` clause will cause *literal-1* or *identifier-1* to be automatically moved into *record-name-1* prior to writing *record-name-1*'s contents to the *file-name-1*. If this clause is not specified, it is the programmer's responsibility to populate *record-name-1* with the desired data prior to executing the `REWRITE`.
4. This statement may not be used with `ORGANIZATION LINE SEQUENTIAL` (see [ORGANIZATION LINE SEQUENTIAL], page 111) files.
5. Rewriting a record does not cause the contents of the file to be physically updated until the next block of the file is read, a `COMMIT` (see [COMMIT], page 300) or `UNLOCK` statement (see [UNLOCK], page 412) is issued or that file is closed.
6. If the file has `ORGANIZATION SEQUENTIAL` (see [ORGANIZATION SEQUENTIAL], page 109):
   A. The record to be rewritten will be the one retrieved by the most-recently executed `READ` (see [READ], page 366) of the file.
   B. If the `FD` of the file contains the `RECORD CONTAINS` or `RECORD IS VARYING` clause, and that clause allows the record size to vary, the size of *record-name-1* cannot be altered.
7. If the file has `ORGANIZATION RELATIVE` (see [ORGANIZATION RELATIVE], page 113) or `ORGANIZATION INDEXED` (see [ORGANIZATION INDEXED], page 115):

A. If the file has `ACCESS MODE SEQUENTIAL`, the record to be rewritten will be the one retrieved by the most-recently executed `READ` of the file. If the file has `ACCESS MODE RANDOM` or `ACCESS MODE DYNAMIC`, no `READ` is required before a record may be rewritten — the `RELATIVE KEY` or `RECORD KEY` definition for the file, respectively, will specify the record to be updated.

B. If the `FD` of the file contains the `RECORD CONTAINS` or `RECORD IS VARYING` clause, and that clause allows the record size to vary, the size *can* be altered.

8. The optional `LOCK` options may be used to manually control access to the re-written record by other programs while this program is running. See [Record Locking], page 59, to review the various record locking behaviours.
9. The optional `INVALID KEY` and `NOT INVALID KEY` clauses may be used to detect and react to the failure or success, respectively, by detecting non-zero (typically 23 = key not found = record not found) and 00 file status codes, respectively. See [File Status Codes], page 107, for additional information.

7.8.37. ROLLBACK

ROLLBACK Syntax

```
ROLLBACK
~~~~~~~~
```

The `ROLLBACK` statement has the same effect as if an `UNLOCK` statement (see [UNLOCK], page 412) were executed against every open file in the program.

1. All locks currently being held for all open files will be released.
2. See [Record Locking], page 59, to review the various record locking behaviours.

7.8.38. SEARCH

SEARCH Syntax

```
SEARCH table-name-1
~~~~~~

   [ VARYING index-name-1 ]
     ~~~~~~~

   [ AT END imperative-statement-1 ]
        ~~~

   { WHEN conditional-expression-1 imperative-statement-2 }...
     ~~~~

[ END-SEARCH ]
  ~~~~~~~~~~
```

The `SEARCH` statement is used to sequentially search a table, stopping either once a specific value is located within the table or when the table has been completely searched.

1. The reserved word `AT` is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. The searching process will be controlled through a *Search Index* — a data item with a `USAGE` (see [USAGE], page 232) of `INDEX`. The search index is either the *index-name-1* identifier specified on the `VARYING` clause or — if no `VARYING` is specified — the `USAGE INDEX` data item implicitly created by an `INDEXED BY` (see [OCCURS], page 190) clause in the table's definition.
3. At the time the `SEARCH` statement is executed, the current value of the search index data item will define the starting position in the table where the searching process will begin. Typically, one initializes that index to a value of 1 before starting the `SEARCH` via `SET` *`search-index`* `TO 1`.
4. Each of the *conditional-expression-n*s on the `WHEN` clause(s) should involve a data element within the table, subscripted using the search index.
5. The searching process is as follows:
   A. Each *conditional-expression-n* will be evaluated, in turn, until either one evaluates to a value of `TRUE` or all have evaluated to `FALSE`.
   B. The *imperative-statement-n* (see [Imperative Statement], page 722) specified on the `WHEN` clause whose *conditional-expression-n* evaluated to `TRUE` will be executed; after that, the search will be considered complete and control will fall into the first executable statement following the `SEARCH`.
   C. If all *conditional-expression-n*s evaluated to FALSE:
      - The search index will be incremented by 1
      - If the search index now has a value greater than the number of entries in the table, the search is considered to have failed and the *imperative-statement-1* on the optional `AT END` clause, if any, will be executed. After that, control will fall into the first executable statement following the `SEARCH`.

- If the search index now has a value less than or equal to the number of entries in the table, search processing returns back to step A.

7.8.39. SEARCH ALL

SEARCH ALL Syntax

```
SEARCH ALL table-name-1
~~~~~~ ~~~

   [ AT END imperative-statement-1 ]
        ~~~

     WHEN conditional-expression-1 imperative-statement-2
     ~~~~

[ END-SEARCH ]
  ~~~~~~~~~~
```

The SEARCH ALL statement performs a binary, or half-interval, search against a sorted table. This is generally *significantly* faster than performing a sequential SEARCH of a table, especially if the table contains a large number of entries.

1. The reserved word AT is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. To be eligible for searching via SEARCH ALL:
   A. The OCCURS clause of *table-name-1* must contain the following elements:
      - An INDEXED BY entry to define an implicit search index data item with a USAGE (see [USAGE], page 232) of INDEX.
      - An ASCENDING KEY or DESCENDING KEY clause to specify the field within the table by which all entries in the table are sorted.

   B. Just because the table has one or more KEY clauses doesn't mean the data is actually *in* that sequence in the table — the actual sequence of the data *must* agree with the KEY clause(s)! A table-based SORT (see [Table SORT], page 395) can prove very useful in this regard.

   C. No two records in the table may have the same KEY field values. If the table has multiple KEY definitions, then no two records in the table may have the same *combination* of KEY field values.
3. If rule *A* is violated, the compiler will reject the SEARCH ALL. If rules *B* and/or *C* are violated, there will be no message issued by the compiler, but the run-time results of a SEARCH ALL against the table will probably be incorrect.
4. The *conditional-expression-1* should involve the KEY field(s), using the search index (the table's INDEXED BY index name) as a subscript.
5. The function of the single, mandatory, WHEN clause is to compare the key field(s) of the table, as indexed by the search index data item, against whatever literal and/or identifier values you are comparing the key field(s) to in the *conditional-expression-1* in order to locate the desired entry in the table. The search index will be automatically varied in a manner designed to require the minimum number of tests.

6. The internal processing of the SEARCH ALL statement begins by setting internal "first" and "last" pointers to the 1st and last entry locations of the table. Processing then proceeds as follows:
   A. The entry half-way between "first" and "last" is identified. We'll call this the "current" entry, and will set its table entry location into *index-name-1*.
   B. The *conditional-expression-1* is evaluated. This comparison of the key(s) against the target literal/identifier values will have one of three possible outcomes:
      - If the key(s) and value(s) match, *imperative-statement-2* (see [Imperative Statement], page 722) is executed, after which control falls through into the next statement following the `SEARCH ALL`.
      - If the key(s) are LESS THAN the value(s), then the table entry being searched for can only occur in the "current" to "last" range of the table, so a new "first" pointer value is set (it will be set to the "current" pointer).
      - If the key(s) are GREATER THAN the value(s), then the table entry being searched for can only occur in the "first" to "current" range of the table, so a new "last" pointer value is set (it will be set to the "current" pointer).
   C. If the new "first" and "last" pointers are different than the old "first" and "last" pointers, there's more left to be searched, so return to step *A* and continue.
   D. If the new "first" and "last" pointers are the same as the old "first" and "last" pointers, the table has been exhausted and the entry being searched for cannot be found; *imperative-statement-1* is executed, after which control falls through into the next statement following the `SEARCH ALL`. If there is no `AT END` clause coded, control simply falls into the next statement following the `SEARCH ALL`.
7. The net effect of the above algorithm is that only a fraction of the number of elements in the table need ever be tested in order to decide whether or not a particular entry exists. This is because the half the remaining entries in the table are discarded each time an entry is checked.
8. Computer scientists will compare the two techniques implemented by the `SEARCH` and `SEARCH ALL` statements as follows:
9. When searching a table with N entries, a sequential search will need an average of $N/2$ tests and a worst case of N tests in order to find an entry and N tests to identify that an entry doesn't exist.
10. When searching a table with N entries, a binary search will need a worst-case of $\log2(N)$ tests in order to find an entry and $\log2(N)$ tests to identify that an entry doesn't exist (N = the number of entries in the table), where *log2* is the base-2 logarithm function.

Here's a more practical view of the difference. Let's say that a table has 1,000 entries in it. With a sequential search, on average, you'll have to check 500 of them to find an entry and you'll have to look at all 1,000 of them to find that an entry doesn't exist.

With a binary search, express the number of entries as a binary number (1,000 = 1111101000), count the number of digits in the result (which is, essentially, what a logarithm is, when rounded up to the next integer — the number of digits a decimal number would have if expressed in the logarithm's number base). In this case, we end up with 10

— *that* is the worst-case number of tests required to find an entry or to identify that it doesn't exist. That's quite an improvement!

7.8.40. SET

7.8.40.1. SET ENVIRONMENT

SET ENVIRONMENT Syntax

```
SET ENVIRONMENT { literal-1    } TO { literal-2    }
~~~ ~~~~~~~~~~~ { identifier-1 } ~~ { identifier-2 }
```

The `SET ENVIRONMENT` statement provides a straight-forward means of setting environment values from within a program.

1. The value of *literal-1* or *identifier-1* specifies the name of the environment variable to set.
2. The value of *literal-2* or *identifier-2* specifies the value to be assigned to the environment variable.
3. Environment variables created or changed from within GnuCOBOL programs will be available to any sub-shell processes spawned by that program (i.e. `CALL "SYSTEM"`) but will not be known to the shell or console window that started the GnuCOBOL program.

This is a much simpler and more readable means of setting environment variables than by using the `DISPLAY UPON ENVIRONMENT-NAME` statement (see [DISPLAY UPON ENVIRONMENT-NAME], page 308). For example, these two code sequences produce identical results:

```
DISPLAY "VARNAME" UPON ENVIRONMENT-NAME
DISPLAY "VALUE" UPON ENVIRONMENT-VALUE

SET ENVIRONMENT "VARNAME" TO "VALUE"
```

7.8.40.2. SET Program-Pointer

SET Program-Pointer Syntax

```
SET program-pointer-1 TO ENTRY { literal-1    }
~~~                   ~~ ~~~~~ { identifier-1 }
```

The SET *Program-Pointer* statement allows you to retrieve the address of a procedure division code module — specifically the PROGRAM-ID, FUNCTION-ID or an entry-point established via the ENTRY statement (see [ENTRY], page 318).

1. The USAGE (see [USAGE], page 232) of *program-pointer-1* must be PROGRAM-POINTER.
2. The *literal-1* or *identifier-1* value specified must name a primary entry-point name (PROGRAM-ID of a subroutine or FUNCTION-ID of a user-defined function) or an alternate entry-point defined via an ENTRY statement within a subprogram.
3. Once the address of a procedure division code area has been acquired in this way, the address could be passed to a subroutine (usually written in C) for whatever use it needs it for. For examples of PROGRAM-POINTERs at work, see the discussions of the CBL_ERROR_PROC built-in system subroutine (see [CBL_ERROR_PROC], page 568) and CBL_EXIT_PROC built-in system subroutine (see [CBL_EXIT_PROC], page 570).

7.8.40.3. SET ADDRESS

SET ADDRESS Syntax

```
SET [ ADDRESS OF ] { pointer-name-1 }...
~~~   ~~~~~~~ ~~   { identifier-1   }

    TO [ ADDRESS OF ]  { pointer-name-2 }
    ~~   ~~~~~~~ ~~    { identifier-2   }
```

The SET ADDRESS statement can be used to work with the addresses of data items rather than their contents.

1. When the ADDRESS OF clause is used *before* the TO you will be using this statement to alter the address of a linkage section or BASED (see [BASED], page 160) data item. Without that clause you will be assigning an address to one or more data items whose USAGE (see [USAGE], page 232) is POINTER.
2. When the ADDRESS OF clause is used *after* the TO, this statement will be identifying the address of *identifier-2* as the address to be assigned to *identifier-1* or stored in *pointer-name-1*.
3. If the ADDRESS OF clause is absent after the TO, the contents of *pointer-name-2* will serve as the address to be assigned.

7.8.40.4. SET Index

SET Index Syntax

```
SET index-name-1 TO { literal-1    }
~~~              ~~ { identifier-2 }
```

This statement assigns a value to a `USAGE INDEX` data item.

1. Either the `USAGE` (see [USAGE], page 232) of *index-name-1* should be `INDEX`, or *index-name-1* must be identified in a table `INDEXED BY` clause.

7.8.40.5. SET UP/DOWN

SET UP/DOWN Syntax

```
SET identifier-1 { UP   } BY [ LENGTH OF ] { literal-1    }
~~~              { ~~   } ~~   ~~~~~~ ~~   { identifier-2 }
                 { DOWN }
                   ~~~~
```

Use this statement to increment or decrement the value of an index or pointer by a specified amount.

1. The `USAGE` (see [USAGE], page 232) of *identifier-1* must be `INDEX`, `POINTER` or `PROGRAM-POINTER`.
2. The typical usage when *identifier-1* is a `USAGE INDEX` data item is to increment its value `UP` or `DOWN` by 1, since an index is usually being used to sequentially walk through the elements of a table.

7.8.40.6. SET Condition Name

SET Condition Name Syntax

```
SET condition-name-1...  TO { TRUE  }
~~~                      ~~ { ~~~~  }
                            { FALSE }
                              ~~~~~
```

The `SET` *`Condition Name`* statement provides one method of specifying the `TRUE` / `FALSE` value of a level-88 condition name.

1. By setting the specified *condition-name-1*(s) to a `TRUE` or `FALSE` value, you will actually be assigning a value to the parent data item(s) to which the condition name data item(s) is(are) subordinate to.
2. When specifying `TRUE`, the value assigned to each parent data item will be the first value specified on the condition name's `VALUE` clause.
3. When specifying `FALSE`, the value assigned to each parent data item will be the value specified for the `FALSE` clause of the condition name's definition; if any *condition-name-1* occurrence lacks a `FALSE` clause, the `SET` statement will be rejected by the compiler.

7.8.40.7. SET Switch

SET Switch Syntax

```
SET mnemonic-name-1...  TO { ON  }
~~~                     ~~ { ~~  }
                           { OFF }
                             ~~~
```

This form of the `SET` statement is used to turn switches on or off.

1. Switches are defined using the `SPECIAL-NAMES` (see [SPECIAL-NAMES], page 92) paragraph.
2. Switches may be tested via the `IF` statement (see [IF], page 338) and a Switch-Status Condition. See [Switch-Status Conditions], page 49, for more information.

7.8.40.8. SET ATTRIBUTE

SET ATTRIBUTE Syntax

```
SET identifier-1 ATTRIBUTE { { BELL          } { ON  }...
~~~              ~~~~~~~~~   { ~~~~          } { ~~  }
                             { BLINK         } { OFF }
                             { ~~~~~         }   ~~~
                             { HIGHLIGHT     }
                             { ~~~~~~~~~     }
                             { LEFTLINE      }
                             { ~~~~~~~~      }
                             { LOWLIGHT      }
                             { ~~~~~~~~      }
                             { OVERLINE      }
                             { ~~~~~~~~      }
                             { REVERSE-VIDEO }
                             { ~~~~~~~~~~~~~ }
                             { UNDERLINE     }
                               ~~~~~~~~~
```

The `SET ATTRIBUTE` statement may be used to modify one or more attributes of a screen section data item at run-time.

1. When making an attribute change to *identifier-1*, the change will not become visible on the screen until the screen section data item containing *identifier-1* is next accepted (if *identifier-1* is an input field) or is next displayed (if *identifier-1* is not an input field).
2. The attributes shown in the syntax diagram are the only ones that may be altered by this statement. See [Data Description Clauses], page 154, for information on their usage.

7.8.40.9. SET LAST EXCEPTION

SET ATTRIBUTE Syntax

```
SET LAST EXCEPTION TO { OFF }
~~~ ~~~~ ~~~~~~~~~ ~~   ~~~
```

The `SET LAST EXCEPTION` statement will set the last program exception status to indicate no exception.

1. The predefined object reference EXCEPTION-OBJECT is set to null, and the last exception status is set to indicate no exception.
2. This action resets the global exception object completely (FUNCTION EXCEPTION-{FILE, LOCATION, STATEMENT, STATUS }), and will not show anything afterwards), no matter what the last exception was (such as a divide by zero). Use with care.

7.8.41. SORT

7.8.41.1. File-Based SORT

File-Based SORT Syntax

```
SORT sort-file-1
~~~~
   { ON { ASCENDING  } KEY identifier-1...  }...
        { ~~~~~~~~~  }
        { DESCENDING }
          ~~~~~~~~~~
   [ WITH DUPLICATES IN ORDER ]
          ~~~~~~~~~~
   [ COLLATING SEQUENCE IS alphabet-name-1 ]
     ~~~~~~~~~

   { INPUT PROCEDURE IS procedure-name-1        }
   { ~~~~~~ ~~~~~~~~~                           }
   {       [ THRU|THROUGH procedure-name-2 ]    }
   {         ~~~~ ~~~~~~~                       }
   { USING file-name-1 ...                      }
     ~~~~~

   { OUTPUT PROCEDURE IS procedure-name-3       }
   { ~~~~~~ ~~~~~~~~~                           }
   {       [ THRU|THROUGH procedure-name-4 ]    }
   {         ~~~~ ~~~~~~~                       }
   { GIVING  file-name-2 ...                    }
     ~~~~~~
```

The `DUPLICATES` clause is syntactically recognized but is otherwise non-functional.

This format of the `SORT` statement is designed to sort large volumes of data according to one or more key fields.

1. The reserved words `IN`, `IS`, `KEY`, `ON`, `ORDER`, `SEQUENCE` and `WITH` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved words `THRU` and `THROUGH` are interchangeable.
3. GnuCOBOL always behaves as if the `WITH DUPLICATES IN ORDER` clause is specified, even if it isn't.

   While any COBOL implementation's sort or merge facilities guarantee that records with duplicate key values will be in proper sequence with regard to other records with different key values, they generally make no promises as to the resulting relative sequence of records having duplicate key values with one another.

Some COBOL implementations provide this optional clause to force their sort and merge facilities to retain duplicate key-value records in their original input sequence, relative to one another.

4. The *sort-file-1* named on the `SORT` statement must be defined using a sort description (`SD` (see [File/Sort-Description], page 124)). This file is referred to in the remainder of this discussion as the *sort work file*.
5. If specified, *file-name-1* and *file-name-2* must reference `ORGANIZATION LINE SEQUENTIAL` (see [ORGANIZATION LINE SEQUENTIAL], page 111) or `ORGANIZATION SEQUENTIAL` (see [ORGANIZATION SEQUENTIAL], page 109) files. These files must be defined using a file description (`FD` (see [File/Sort-Description], page 124)). The same file(s) may be used for *file-name-1* and *file-name-2*.
6. The *identifier-1* ... field(s) must be defined as field(s) within a record of *sort-file-1*.
7. A sort work file is never opened or closed.
8. The sorting process works in three stages — the Input Stage, the Sort Stage and the Output Stage.
9. The following points pertain to the Input Stage:
   A. The data to be sorted is loaded into the sort work file either by copying the entire contents of the file(s) named on the `USING` clause (done automatically by the sort) or by utilizing an input procedure.
   B. When `USING` is specified, none of the *file-name-1* files may be open at the time the `SORT` statement is executed.
   C. When an input procedure is used, the procedure(s) specified on the `INPUT PROCEDURE` clause will be invoked as if by a procedural `PERFORM` statement (see [Procedural PERFORM], page 360) with no `VARYING`, `TIMES` or `UNTIL` options specified. Records will be loaded into the sort work file — one at a time — within the input procedure using the `RELEASE` statement (see [RELEASE], page 371). This, by the way, is how you could sort the contents of relative or indexed files.

      A `GO TO` statement (see [GO TO], page 335) that transfers control out of the input procedure will terminate the `SORT` statement but allows the program to continue executing from the point where the `GO TO` statement transferred control to. Once an input procedure has been "aborted" using a `GO TO` it cannot be resumed, and the contents of the sort work file are lost. You may, however, re-execute the `SORT` statement itself.[1]

      An input procedure should be terminated in the same way a procedural `PERFORM` statement would be.

      Neither a another file-based `SORT` statement nor a `MERGE` statement may be executed within the input procedure unless those statements utilize a different sort or merge work file.
   D. Once the input procedure terminates, the input phase is complete.
   E. As data is loaded into the sort work file, it is actually being buffered in dynamically-allocated memory. Only if the amount of data to be sorted exceeds the amount

[1] Using a `GO TO` statement to prematurely terminate a sort, or re-starting a previously-cancelled sort is not considered good programming style and should be avoided.

of available sort memory (128 MB) will actual disk files be allocated and utilized. There is a COB_SORT_MEMORY run-time environment variable (see [Run Time Environment Variables], page 658) that you may use to allocate more or less memory to the sorting process.

10. The following points pertain to the Sort Stage:

    A. The sort will take place by arranging the data records in the sequence defined by the KEY specification(s) on the SORT statement according to the COLLATING SEQUENCE specified on the SORT (if any) or — if none was defined — the PROGRAM COLLATING SEQUENCE (see [OBJECT-COMPUTER], page 90). Keys may be any supported data type and USAGE (see [USAGE], page 232) except for level-78 or level-88 data items.

    B. For example, let's assume we're sorting a series of financial transactions. The SORT statement might look like this:

```
SORT Sort-File
    ASCENDING  KEY Transaction-Date
    ASCENDING  KEY Account-Number
    DESCENDING KEY Transaction-Amount
```

    The effect of this statement will be to sort all transactions into ascending order of the date the transaction took place (oldest first, newest last). Unless the business running this program is going out of business, there are most-likely many transactions for any given date. Therefore, within each grouping of transactions all with the same date, transactions will be sub-sorted into ascending sequence of the account number the transactions apply to. Since it's quite possible there might be multiple transactions for an account on any given date, a third level sub-sort will arrange all transactions for the same account on the same date into descending sequence of the actual amount of the transaction (largest first, smallest last). If two or more transactions of $100.00 were recorded for account #12345 on the 31st of August 2009, those transactions will be retained in the order in which they were loaded into the sort work file.

    C. Should disk work files be necessary due to the amount of data being sorted, they will be automatically allocated to disk in a folder defined by the TMPDIR run-time environment variable, TMP run-time environment variable or TEMP run-time environment variable run-time environment variables (see [Run Time Environment Variables], page 658) (checked for existence in that sequence). These disk files will be automatically purged upon SORT termination or program execution termination (normal or otherwise).

11. The following points pertain to the Output Stage:

    A. Once the sort stage is complete, a copy of the sorted data will be written to each *file-name-2* if the GIVING clause was specified. None of the *file-name-2* files can be open at the time the sort is executed.

    B. When an output procedure is used, the procedure(s) specified on the OUTPUT PROCEDURE clause will be invoked as if by a procedural PERFORM statement (see [Procedural PERFORM], page 360) with no VARYING, TIMES or UNTIL options spec-

ified. Records will be retrieved from the sort work file — one at a time — within the output procedure using the `RETURN` statement (see [RETURN], page 373).

A `GO TO` statement (see [GO TO], page 335) that transfers control out of the output procedure will terminate the `SORT` statement but allows the program to continue executing from the point where the `GO TO` statement transferred control to. Once an output procedure has been "aborted" using a `GO TO` it cannot be resumed, and the contents of the sort work file are lost. You may, however, re-execute the `SORT` statement itself. USING A `GO TO` statement[2]

An output procedure should be terminated in the same way a procedural `PERFORM` statement would be.

Neither a another file-based `SORT` statement nor a `MERGE` statement may be executed within the output procedure unless those statements utilize a different sort or merge work file.

C. Once the output procedure terminates, the sort is complete.

[2] To prematurely terminate a sort, or re-starting a previously-cancelled sort is not considered good programming style and should be avoided.

7.8.41.2. Table SORT

Table SORT Syntax

```
SORT table-name-1
~~~~
   { ON { ASCENDING  } KEY identifier-1...  }...
        { ~~~~~~~~~  }
        { DESCENDING }
          ~~~~~~~~~~

   [ WITH DUPLICATES IN ORDER ]
          ~~~~~~~~~~

   [ COLLATING SEQUENCE IS alphabet-name-1 ]
     ~~~~~~~~~
```

The `DUPLICATES` clause is syntactically recognized but is otherwise non-functional.

This format of the `SORT` statement sorts relatively small quantities of data — namely data contained in a data division table — according to one or more key fields.

1. The reserved words `IN`, `IS`, `KEY`, `ON`, `ORDER`, `SEQUENCE` and `WITH` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. GnuCOBOL always behaves as if the `WITH DUPLICATES IN ORDER` clause is specified, even if it isn't.

   While any COBOL implementation's sort or merge facilities guarantee that records with duplicate key values will be in proper sequence with regard to other records with different key values, they generally make no promises as to the resulting relative sequence of records having duplicate key values with one another.

   Some COBOL implementations provide this optional clause to force their sort and merge facilities to retain duplicate key-value records in their original input sequence, relative to one another.
3. The *table-name-1* data item must be a table defined in any data division section *except* the report or screen sections.
4. The data within *table-name-1* will be sorted in-place (i.e. no sort file is required).
5. The sort will take place by rearranging the data in *table-name-1* into the sequence defined by the `KEY` specification(s) on the `SORT` statement, according to the `COLLATING SEQUENCE` specified on the `SORT` (if any) or — if none was defined — the `PROGRAM COLLATING SEQUENCE` (see [OBJECT-COMPUTER], page 90). Keys may be any supported data type and `USAGE` (see [USAGE], page 232) except for level-78 or level-88 data items.
6. If you are sorting *table-name-1* for the purpose of preparing the table for use with a `SEARCH ALL` statement (see [SEARCH ALL], page 379), care must be taken that the `KEY` specifications on the `SORT` agree with those in the table's definition.

7. Although the specification of one or more `KEY` clauses is optional, currently, a table sort with no `KEY` specification(s) made on the `SORT` statement is unsupported by GnuCOBOL and will be rejected by the compiler.

7.8.42. START

START Syntax

```
START file-name-1
~~~~~
  [ { FIRST                                                    } ]
    { ~~~~~                                                    }
    { LAST                                                     }
    { ~~~~                                                     }
    { KEY { IS EQUAL TO | IS = | EQUALS         } identifier-1 }
      ~~~ {    ~~~~~             ~~~~~~         }
          { IS GREATER THAN | IS >               }
          {    ~~~~~~~                           }
          { IS GREATER THAN OR EQUAL TO | IS >= }
          {    ~~~~~~~      ~~ ~~~~~             }
          { IS NOT LESS THAN                     }
          {    ~~~ ~~~~                          }
          { IS LESS THAN | IS <                  }
          {    ~~~~                              }
          { IS LESS THAN OR EQUAL TO | IS <=     }
          {    ~~~~      ~~ ~~~~~               }
          { IS NOT GREATER THAN                  }
              ~~~ ~~~~~~~


      [ WITH {SIZE}   arithmetic-expression ]
             ~~~~
      [      {LENGTH} arithmetic-expression ]
             ~~~~~~


   [ INVALID KEY imperative-statement-1 ]
     ~~~~~~~
   [ NOT INVALID KEY imperative-statement-2 ]
     ~~~ ~~~~~~~
[ END-START ]
  ~~~~~~~~~
```

The `START` statement defines the logical starting point within a relative or indexed file for subsequent sequential read operations. It positions an internal logical record pointer to a particular record in the file, but does not actually transfer any of that record's data into the record buffer.

1. The reserved words `IS`, `THAN` and `TO` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. To use this statement, *file-name-1* must be an `ORGANIZATION RELATIVE` (see

[ORGANIZATION RELATIVE], page 113) or `ORGANIZATION INDEXED` (see [ORGANIZATION INDEXED], page 115) file that must have been defined with an `ACCESS MODE DYNAMIC` or `ACCESS MODE SEQUENTIAL` in its `SELECT` statement (see [SELECT], page 104).

3. At the time this statement is executed, *file-name-1* must be open in either `INPUT` or `I-O` (see [File OPEN Modes], page 359) mode.
4. If *file-name-1* is a relative file, *identifier-1* must be the defined `RELATIVE KEY` of the file.
5. If *file-name-1* is an indexed file, *identifier-1* must be the defined `RECORD KEY` of the file or any of the `ALTERNATE RECORD KEY` fields for the file.
6. If no `FIRST`, `LAST` or `KEY` clause is specified, `KEY IS EQUAL TO` *xxx* will be assumed, where *xxx* is the defined `RELATIVE KEY` of (if *file-name-1* is a relative file) or the defined `RECORD KEY` (if *file-name-1* is an indexed file).
7. After successful execution of a `START` statement, the internal logical record pointer into the *file-name-1* data will be positioned to the record which satisfied the actual or implied `FIRST`, `LAST` or `KEY` clause specification, as follows:

   `FIRST` the logical record pointer will point to the first record in the file.

   `LAST` the logical record pointer will point to the last record in the file.

   `KEY` (specified or implied), and the relation used is. Warning: Later versions of the compiler may well not use implied, so always specify it and it makes the code easier to read any way.

   `SIZE` WITH `SIZE` or `LENGTH` arithmetic-expression specifies the number of characters in the key to be used in the positioning process.

   `LENGTH` WITH `LENGTH` or `SIZE` arithmetic-expression specifies the number of characters in the key to be used in the positioning process.

   `SIZE` and `LENGTH` are interchangeable and mean exactly the same.

   `EQUAL TO`, `GREATER THAN` or `GREATER THAN OR EQUAL TO` (or equivalent)
   the logical record pointer will be specified to the *first* record satisfying the relation condition; to identify this record. The file's contents are searched in a first-to-last (in sequence of the key implied by the `KEY` clause), provided the relation is

   `LESS THAN`, `LESS THAN OR EQUAL TO` or `NOT GREATER THAN` (or equivalent)
   the logical record pointer will be specified to the *last* record satisfying the relation condition; to identify this record. The file's contents are searched in a last-to-first (in sequence of the key implied by the `KEY` clause)

   The next sequential `READ` statement will read the record that is pointed to by the logical record pointer.
8. The optional `INVALID KEY` and `NOT INVALID KEY` clauses may be used to detect and react to the failure or success, respectively, by detecting non-zero (typically 23 = key not found = record not found) and 00 file status codes, respectively. See [File Status Codes], page 107, for additional information.

7.8.43. STOP

STOP Syntax

```
STOP { RUN [ { RETURNING|GIVING { literal-1    }           } ] }
~~~~ { ~~~   { ~~~~~~~~~~ ~~~~~~ { identifier-1 }           }   }
     {       {                                              }   }
     {       { WITH { ERROR  } STATUS [ { literal-2    } ] }   }
     {       {      { ~~~~~  }          { identifier-2 }   }   }
     {       {      { NORMAL }                             }   }
     {                ~~~~~~                                   }
     { literal-3                                              }
```

The `STOP` statement suspends program execution. Some options will allow program execution to resume while others return control to the operating system.

1. The reserved words `STATUS` and `WITH` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved words `RETURNING` and `GIVING` are interchangeable.
3. The `RUN` clause halts the program without displaying any special message to that effect.
4. The *literal-3* clause displays the specified text on the `SYSOUT`/`STDOUT` device, waits for the user to press the Enter key and then — once the key has been pressed — allows the program to continue execution.
5. The optional `RETURNING` clause provides the opportunity to return a numeric value to the operating system (an *exit status*). The manner in which the exit status may be interrogated by the operating system varies, but Windows can use `%ERRORLEVEL%` to query the exit status while Unix shells such as sh, bash and ksh can query the exit status as `$?`. Other Unix shells may have different ways to access return code values.
6. The `STATUS` clause provides another means of returning an exit status. Using the `STATUS` clause is functionally equivalent to using the `RETURNING` clause.
7. Using the `STATUS` clause without a *literal-2* or *identifier-2* will return an exit status of 0 if the `NORMAL` keyword is used or a 1 if `ERROR` was specified.
8. Your program will *always* return an exit status, even if no `RETURNING` or `STATUS` clause is specified. In the absence of the use of these clauses, the value in the `RETURN-CODE` special register (see [Special Registers], page 265) at the time the `STOP` statement is executed will be used as the exit status.
9. Any programmer-defined exit procedure (established via the `CBL_EXIT_PROC` built-in system subroutine (see [CBL_EXIT_PROC], page 570)) will be executed by `STOP RUN`, but not by `STOP` *`literal-3`*.
10. Valid return code values can be in the range -2147483648 to +2147483647.
11. The three code snippets below are all equivalent. They show different ways in which a GnuCOBOL program may be coded to pass an exit status value of 16 back to the operating system and then halt.

1.

```
STOP RUN RETURNING 16
```

2.

```
MOVE 16 TO RETURN-CODE
STOP RUN
```

3.

```
STOP RUN WITH ERROR STATUS 16
```

7.8.44. STRING

STRING Syntax

```
STRING
~~~~~~
   { { literal-1    } [ DELIMITED BY { SIZE         } ] }...
     { identifier-1 }   ~~~~~~~~~    { ~~~~         }
                                     { literal-2    }
                                     { identifier-2 }
     INTO identifier-3
     ~~~~
   [ WITH POINTER identifier-4 ]
         ~~~~~~~
   [ ON OVERFLOW imperative-statement-1 ]
       ~~~~~~~~
   [ NOT ON OVERFLOW imperative-statement-2 ]
     ~~~    ~~~~~~~~
[ END-STRING ]
  ~~~~~~~~~~
```

The `STRING` statement is used to concatenate all or a part of one or more strings together, forming a new string.

1. The reserved words `BY`, `ON` and `WITH` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. All literals and identifiers (except for *identifier-4*) must be explicitly or implicitly defined with a `USAGE` (see [USAGE], page 232) of `DISPLAY`. Any of the identifiers may be group items.
3. The `POINTER` data item — *identifier-4* — must be a non-edited elementary integer numeric data item with a value greater than zero.
4. Each *literal-1* / *identifier-1* will be referred to as a source item. The receiving data item is *identifier-3*.
5. The `STRING` statement's processing is based upon a *current character pointer*. The initial value of the current character pointer will be the value of *identifier-4* at the time the `STRING` statement began execution. If no `POINTER` clause is coded, a value of 1 (meaning "the 1st character position") will be assumed for the current character pointer's initial value.
6. For each source item, the contents of the sending item will be copied — character-by-character — into *identifier-3* at the character position specified by the current character pointer. After each character is copied, the current character pointer will be incremented by 1 so that it points to the position within *identifier-3* where the *next* character should be copied.

7. The `DELIMITED BY` clause specifies how much of each source item will be copied into *identifier-3*. `DELIMITED BY SIZE` (the default if no `DELIMITED BY` clause is specified) causes the *entire* contents of the source item to be copied into *identifier-3*.
8. Using `DELIMITED BY` *`literal-2`* or `DELIMITED BY` *`identifier-2`* causes only the contents of the source item up to but not including the character sequence specified by the literal or identifier to be copied.
9. `STRING` processing will cease when one of the following occurs:
   A. The initial value of the current character pointer is less than 1 or greater than the number of characters in *identifier-3*, or. . .
   B. The value of the current character pointer exceeds the size of *identifier-3* at the point the STRING statement wants to copy a character into *identifier-3*, or. . .
   C. All sending items have been fully processed
10. If event *A* occurs, *identifier-3* will remain unchanged.
11. The occurrence of either event *A* or *B* triggers what is referred to as an *overflow condition*.
12. The *identifier-3*) is neither automatically initialized (to spaces or any other value) at the start of a `STRING` statement nor will it be space-filled should the total number of sending item characters copied into it be less than its size. You may explicitly initialize *identifier-3* yourself via the `INITIALIZE` (see [INITIALIZE], page 339) or `MOVE` (see [MOVE], page 352) statements before executing the `STRING` if you wish.
13. The optional `ON OVERFLOW` and `NOT ON OVERFLOW` clauses may be used to detect and react to the occurrence or not, respectively, of an overflow condition. See [ON OVERFLOW + NOT ON OVERFLOW], page 260, for additional information.

7.8.45. SUBTRACT

7.8.45.1. SUBTRACT FROM

SUBTRACT FROM Syntax

```
SUBTRACT { literal-1    }...  FROM { identifier-2
~~~~~~~~ { identifier-1 }     ~~~~

       [ ROUNDED [ MODE IS { AWAY-FROM-ZERO         } ] ] }...
         ~~~~~~~   ~~~~    { ~~~~~~~~~~~~~~         }
                           { NEAREST-AWAY-FROM-ZERO }
                           { ~~~~~~~~~~~~~~~~~~~~~~ }
                           { NEAREST-EVEN           }
                           { ~~~~~~~~~~~~           }
                           { NEAREST-TOWARD-ZERO    }
                           { ~~~~~~~~~~~~~~~~~~     }
                           { PROHIBITED             }
                           { ~~~~~~~~~~             }
                           { TOWARD-GREATER         }
                           { ~~~~~~~~~~~~~~         }
                           { TOWARD-LESSER          }
                           { ~~~~~~~~~~~~~          }
                           { TRUNCATION             }
                             ~~~~~~~~~~

    [ ON SIZE ERROR imperative-statement-1 ]
         ~~~~ ~~~~~

    [ NOT ON SIZE ERROR imperative-statement-2 ]
      ~~~    ~~~~ ~~~~~

[ END-SUBTRACT ]
  ~~~~~~~~~~~~
```

This format of the `SUBTRACT` statement generates the arithmetic sum of all arguments that appear before the `FROM` (*identifier-1* or *literal-1*) and subtracts that sum from each *identifier-2*.

1. The reserved words `IS` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. Both *identifier-1* and *identifier-2* must be numeric unedited data items.
3. *literal-1* must be a numeric literal.
4. The optional `ROUNDED` (see [ROUNDED], page 261) clause available to each *identifier-2* will control how non-integer results will be saved.
5. The optional `ON SIZE ERROR` and `NOT ON SIZE ERROR` clauses may be used to detect and react to the failure or success, respectively, of an attempt to perform a calculation. In

this case, failure is defined as being an *identifier-2* with an insufficient number of digit positions available to the left of any implied decimal point. See [ON SIZE ERROR + NOT ON SIZE ERROR], page 261, for additional information.

7.8.45.2. SUBTRACT GIVING

SUBTRACT GIVING Syntax

```
SUBTRACT { literal-1    }...  FROM identifier-2
~~~~~~~~ { identifier-1 }     ~~~~

    GIVING { identifier-3
    ~~~~~~
       [ ROUNDED [ MODE IS { AWAY-FROM-ZERO         } ] ] }...
         ~~~~~~~   ~~~~    { ~~~~~~~~~~~~~~         }
                           { NEAREST-AWAY-FROM-ZERO }
                           { ~~~~~~~~~~~~~~~~~~~~~~ }
                           { NEAREST-EVEN           }
                           { ~~~~~~~~~~~~           }
                           { NEAREST-TOWARD-ZERO    }
                           { ~~~~~~~~~~~~~~~~~~     }
                           { PROHIBITED             }
                           { ~~~~~~~~~~             }
                           { TOWARD-GREATER         }
                           { ~~~~~~~~~~~~~~         }
                           { TOWARD-LESSER          }
                           { ~~~~~~~~~~~~~          }
                           { TRUNCATION             }
                             ~~~~~~~~~~

  [ ON SIZE ERROR imperative-statement-1 ]
      ~~~~ ~~~~~

  [ NOT ON SIZE ERROR imperative-statement-2 ]
    ~~~    ~~~~ ~~~~~

[ END-SUBTRACT ]
  ~~~~~~~~~~~~
```

The `SUBTRACT GIVING` statement generates the arithmetic sum of all arguments that appear before the `FROM` (*identifier-1* or *literal-1*), subtracts that sum from the contents of *identifier-2* and then replaces the contents of the identifiers listed after the `GIVING` (*identifier-3*) with that result.

1. The reserved words `IS` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. Both *identifier-1* and *identifier-2* must be numeric unedited data items.
3. *literal-1* must be a numeric literal.
4. *identifier-3* must be a numeric (edited or unedited) data item.
5. The optional `ROUNDED` (see [ROUNDED], page 261) clause available to each *identifier-2* will control how non-integer results will be saved.

6. The optional `ON SIZE ERROR` and `NOT ON SIZE ERROR` clauses may be used to detect and react to the failure or success, respectively, of an attempt to perform a calculation. In this case, failure is defined as being an *identifier-2* with an insufficient number of digit positions available to the left of any implied decimal point. See [ON SIZE ERROR + NOT ON SIZE ERROR], page 261, for additional information.

7.8.45.3. SUBTRACT CORRESPONDING

SUBTRACT CORRESPONDING Syntax

```
SUBTRACT CORRESPONDING identifier-1 FROM identifier-2
~~~~~~~~               ~~~~~~~~~~~~ ~~~~
  [ ROUNDED [ MODE IS { AWAY-FROM-ZERO         } ] ]
    ~~~~~~~   ~~~~    { ~~~~~~~~~~~~~~         }
                      { NEAREST-AWAY-FROM-ZERO }
                      { ~~~~~~~~~~~~~~~~~~~~~~ }
                      { NEAREST-EVEN           }
                      { ~~~~~~~~~~~~           }
                      { NEAREST-TOWARD-ZERO    }
                      { ~~~~~~~~~~~~~~~~~~     }
                      { PROHIBITED             }
                      { ~~~~~~~~~~             }
                      { TOWARD-GREATER         }
                      { ~~~~~~~~~~~~~~         }
                      { TOWARD-LESSER          }
                      { ~~~~~~~~~~~~~          }
                      { TRUNCATION             }
                        ~~~~~~~~~~
  [ ON SIZE ERROR imperative-statement-1 ]
       ~~~~ ~~~~~
  [ NOT ON SIZE ERROR imperative-statement-2 ]
    ~~~    ~~~~ ~~~~~
[ END-SUBTRACT ]
  ~~~~~~~~~~~~
```

The `SUBTRACT CORRESPONDING` statement generates code equivalent to individual `SUBTRACT FROM` statements for corresponding matches of data items found subordinate to the two identifiers.

1. The reserved words `IS` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. Both *identifier-1* and *identifier-2* must be group items.
3. See [CORRESPONDING], page 258, for information on how corresponding matches will be found between *identifier-1* and *identifier-2*.
4. The optional `ROUNDED` (see [ROUNDED], page 261) clause available to each *identifier-2* will control how non-integer results will be saved.
5. The optional `ON SIZE ERROR` and `NOT ON SIZE ERROR` clauses may be used to detect and react to the failure or success, respectively, of an attempt to perform a calculation. In this case, failure is defined as being an *identifier-2* with an insufficient number of digit

positions available to the left of any implied decimal point. See [ON SIZE ERROR + NOT ON SIZE ERROR], page 261, for additional information.

7.8.46. SUPPRESS

SUPPRESS Syntax

```
SUPPRESS PRINTING
~~~~~~~~
```

The SUPPRESS statement causes the presentation of a report group to be suppressed.

1. The reserved word PRINTING is optional and may be omitted. The presence or absence of this word has no effect upon the program.
2. This statement may only appear within a USE BEFORE REPORTING procedure (in DECLARATIVES (see [DECLARATIVES], page 254)).
3. SUPPRESS only prevents the presentation of the report group within whose USE BEFORE REPORTING procedure the statement occurs.
4. This statement must be executed each time presentation of the report group is to be suppressed.
5. When a report group's presentation is suppressed, none of the following operations for the report will take place:
   A. Actual presentation of the report group in question.
   B. Processing of any LINE (see [LINE], page 183) clauses within the report group in question.
   C. Processing of the NEXT GROUP (see [NEXT GROUP], page 187) clause (if any) within the report group in question.
   D. Any modification to the LINE-COUNTER special register (see [Special Registers], page 265).
   E. Any modification to the PAGE-COUNTER special register.

7.8.47. TERMINATE

TERMINATE Syntax

```
TERMINATE report-name-1...
~~~~~~~~~
```

The `TERMINATE` statement causes the processing of the specified report(s) to be completed.

1. Each *report-name-1* must be the name of a report having an `RD` (see [REPORT SECTION], page 135) defined for it.
2. The specified report name(s) must be currently initiated (via `INITIATE` (see [INITIATE], page 343)) and cannot yet have been terminated.
3. The `TERMINATE` statement will present each `CONTROL FOOTING` (if any), in reverse sequence of the control hierarchy, starting with the most minor up to `FINAL` (if any). During the presentation of these groups and the processing of any `USE BEFORE REPORTING` procedures for those groups, the prior set of control data item values will be available, as though a control break had been detected at the most major control data name.
4. During the presentation of the `CONTROL FOOTING` groups, any necessary `PAGE FOOTING` and `PAGE HEADING` groups will be presented as well.
5. Finally,the `REPORT FOOTING` group, if any, will be presented.
6. If an `INITIATE` is followed by a `TERMINATE` with no intervening `GENERATE` (see [GENERATE], page 332) statements (all pertaining to the same report, of course), no report groups will be presented to the output file.

7.8.48. TRANSFORM

TRANSFORM Syntax

```
TRANSFORM identifier-1 CHARACTERS FROM { literal-1    } TO { literal-2    }
~~~~~~~~~                         ~~~~ { identifier-2 } ~~ { identifier-3 }
```

The `TRANSFORM` statement scans a data item performing a series of mono-alphabetic substitutions, defined by the arguments before and after the `TO` clause.

1. Both *literal-1* and/or *literal-2* must be alphanumeric literals.
2. All of *identifier-1*, *identifier-2* and *identifier-3* must either be group items or alphanumeric data items. Numeric data items with a `USAGE` (see [USAGE], page 232) of `DISPLAY` are accepted, but will generate warning messages from the compiler.
3. The `TRANSFORM` statement will replace characters within *identifier-1* that are found in the string specified *before* the `TO` keyword with the corresponding characters from the string specified *after* the `TO` keyword.
4. Usage of word CHARACTERS has no effect on operations other than for appearance.
5. This statement exists within GnuCOBOL to provide compatibility with COBOL programs written to pre-1985 standards. The `TRANSFORM` statement was made obsolete in the 1985 standard of COBOL, having been replaced by the `CONVERTING` clause of the `INSPECT` statement (see [INSPECT], page 344). New programs should be coded to use `INSPECT CONVERTING` rather than `TRANSFORM`.

7.8.49. UNLOCK

UNLOCK Syntax

```
UNLOCK filename-1 RECORD|RECORDS
~~~~~~
```

This statement synchronizes any as-yet unwritten file I/O buffers to the specified file (if any) and releases any record locks held for records belonging to *file-name-1*.

1. The reserved words `RECORD` and `RECORDS` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. If *file-name-1* is a Sort/Merge work file, no action will be taken.
3. Not all GnuCOBOL implementations support locking. Whether they do or not depends upon the operating system they were built for and the build options that were used when GnuCOBOL was generated. When a program using one of those GnuCOBOL implementations issues an UNLOCK, it will ignored. There will be no compiler message issued. Buffer syncing, if needed, will still occur.
4. For GnuCOBOL UNLOCK is implied in GnuCOBOL on file close so there's no use to do it afterwards. A CLOSE will always trigger syncing the file to disk, too.
5. Doing UNLOCK before a close, will explicit unlock any records with a lock when running on INDEXED files, for other files it will release any locks on the file if it wasn't opened for exclusive locking and will trigger syncing to disk (not done for any INDEXED file).
6. When using Linux and for that matter most *nix platforms, the system maintains it's own cache and buffers for file processing so there can and most likely will be a short delay before all data is written out to disk.
7. See [Record Locking], page 59, for additional information on record locking.

7.8.50. UNSTRING

UNSTRING Syntax

```
UNSTRING identifier-1
~~~~~~~~
     DELIMITED BY { [ ALL ] literal-1 } [ OR { [ ALL ] literal-2 } ]...
     ~~~~~~~~~    {   ~~~             }   ~~ {   ~~~             }
                  { identifier-2      }      { identifier-3      }

     INTO { identifier-4
     ~~~~ [ DELIMITER IN identifier-5 ] [ COUNT IN identifier-6 ] }...
            ~~~~~~~~~                     ~~~~~

   [ WITH POINTER identifier-7 ]
          ~~~~~~~

   [ TALLYING IN identifier-8 ]
     ~~~~~~~~

   [ ON OVERFLOW imperative-statement-1 ]
       ~~~~~~~~

   [ NOT ON OVERFLOW imperative-statement-2 ]
     ~~~    ~~~~~~~~

[ END-UNSTRING ]
  ~~~~~~~~~~~~
```

The `UNSTRING` statement parses a string, extracting any number of sub strings from it.

1. The reserved words `BY`, `IN` and `ON` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. *identifier-1* through *identifier-5* must be explicitly or implicitly defined with a `USAGE` (see [USAGE], page 232) of `DISPLAY`. Any of those identifiers may be group items.
3. Both *literal-1* and *literal-2* must be alphanumeric literals.
4. Each of *identifier-6*, *identifier-7* and *identifier-8* must be elementary non-edited integer numeric items.
5. At the time the `UNSTRING` statement begins execution, *identifier-7* must have a value greater than 0.
6. *identifier-1* will be referred to as the *source string* and each *identifier-4* will be referred to as a *destination field* in the following discussions.
7. The `UNSTRING` statement's processing is based upon a *current character pointer*, the initial value of which will be the value of *identifier-7* at the time the `UNSTRING` statement began execution. If no `POINTER` clause is coded, a value of 1 (meaning "the 1st character position") will be assumed for the current character pointer's initial value.
8. The source string will be parsed into sub strings starting from the current character pointer position. Sub strings are identified by using the various delimiter strings specified on the `DELIMITED BY` clause as inter-sub string separators.

9. Using the `ALL` option allows a delimiter sequence to be an arbitrarily long sequence of occurrences of the delimiter literal whereas its absence treats each occurrence as a separate delimiter. When multiple delimiters are specified, they will be looked for in the source string in the sequence in which they are coded.
10. Two consecutive delimiter sequences will identify a null sub string.
11. Identified sub strings will be moved into each destination field in the sequence they are identified; values moved into a destination field will be truncated if the sub string length exceeds the destination field length, or padded with spaces if the destination field length exceeds the sub string length. Both truncation and padding will be controlled by the presence or absence of a `JUSTIFIED` (see [JUSTIFIED], page 179) clause on the destination field.
12. Each destination field may have an optional `DELIMITER` clause. If a `DELIMITER` clause is specified, *identifier-5* will have the delimiter character string used to identify the sub string for the destination field moved into it. If a destination field was not altered (because an insufficient number of sub strings were identified), *identifier-5* for that destination field will also be unchanged.
13. Each destination field may have an optional `COUNT` clause. If a `COUNT` clause is specified, *identifier-6* will have the size of the sub string (in characters) for the destination field moved into it. If a destination field was not altered (because an insufficient number of sub strings were identified), *identifier-6* for that destination field will also be unchanged.
14. If a `TALLYING` clause is coded, *identifier-8* will be incremented by 1 each time a destination field is populated.
15. None of *identifier-4*, *identifier-5*, *identifier-6*, *identifier-7* or *identifier-8* are initialized by the `UNSTRING` statement. You need to do that yourself via a `MOVE` (see [MOVE], page 352) or `INITIALIZE` statement (see [INITIALIZE], page 339).
16. `UNSTRING` processing will cease when one of the following occurs:
    A. The initial value of the current character pointer is less than 1 or greater than the number of character positions in *identifier-1*, or...
    B. All destination fields have been fully processed
17. If event *A* occurs, none of the destination field contents (or the contents of their `DELIMITER` or *COUNT* identifiers) will be changed.
18. An *overflow* condition exists if either event *A* occurs, or if event *B* occurs with at least one character position in *identifier-1* not having been processed.
19. The optional `ON OVERFLOW` and `NOT ON OVERFLOW` clauses may be used to detect and react to the occurrence or not, respectively, of an overflow condition. See [ON OVERFLOW + NOT ON OVERFLOW], page 260, for additional information.

The following sample program illustrates the `UNSTRING` statement statement.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DEMOUNSTRING.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Full-Name                  PIC X(40).
01  Parsed-Info.
```

```
    05 Last-Name               PIC X(15).
    05 First-Name              PIC X(15).
    05 MI                      PIC X(1).
    05 Delim-LN                PIC X(1).
    05 Delim-FN                PIC X(1).
    05 Delim-MI                PIC X(1).
    05 Count-LN                BINARY-CHAR.
    05 Count-FN                BINARY-CHAR.
    05 Count-MI                BINARY-CHAR.
    05 Tallying-Ctr            BINARY-CHAR.
PROCEDURE DIVISION.
P1. PERFORM UNTIL EXIT
      DISPLAY "Enter Full Name (null quits):"
          WITH NO ADVANCING
      ACCEPT Full-Name
      IF Full-Name = SPACES
        EXIT PERFORM
      END-IF
      INITIALIZE Parsed-Info
      UNSTRING Full-Name
        DELIMITED BY ", "
                  OR ","
                  OR ALL SPACES
        INTO Last-Name
                  DELIMITER IN Delim-LN
                  COUNT IN Count-LN
             First-Name
                  DELIMITER IN Delim-FN
                  COUNT IN Count-FN
             MI
                  DELIMITER IN Delim-MI
                  COUNT IN Count-MI
        TALLYING Tallying-Ctr
    DISPLAY "First-Name=" First-Name
            " Delim='"    Delim-FN
            "' Count="    Count-FN
    DISPLAY "MI        =" MI "              "
            " Delim='"    Delim-MI
            "' Count="    Count-MI
    DISPLAY "Last-Name =" Last-Name
            " Delim='"    Delim-LN
            "' Count="    Count-LN
    DISPLAY "Tally=     " Tallying-Ctr
  END-PERFORM
  DISPLAY "Bye!"
  STOP RUN   .
```

The following is sample output from the program:

```
Enter Full Name (null quits):Cutler, Gary L
First-Name=Gary            Delim=' ' Count=+004
MI        =L               Delim=' ' Count=+001
Last-Name =Cutler          Delim=',' Count=+006
Tally=     +003
Enter Full Name (null quits):Snoddgrass,Throckmorton,P
First-Name=Throckmorton    Delim=',' Count=+012
MI        =P               Delim=' ' Count=+001
Last-Name =Snoddgrass      Delim=',' Count=+010
Tally=     +003
Enter Full Name (null quits):Munster   Herman
First-Name=Herman          Delim=' ' Count=+006
MI        =                Delim=' ' Count=+000
Last-Name =Munster         Delim=' ' Count=+007
Tally=     +002
Enter Full Name (null quits):
Bye!
```

7.8.51. WRITE

WRITE Syntax

```
WRITE record-name-1
~~~~~
   [ FROM { literal-1    } ]
     ~~~~ { identifier-1 }

   [ WITH [ NO ] LOCK ]
            ~~   ~~~~

   [ { BEFORE } ADVANCING { { literal-2    } LINE|LINES } ]
     { ~~~~~~ }           { { identifier-2              }
     { AFTER  }           { PAGE                        }
       ~~~~~              { ~~~~                        }
                          { mnemonic-name-1             }

   [ AT END-OF-PAGE|EOP imperative-statement-1 ]
      ~~~~~~~~~~~ ~~~

   [ NOT AT END-OF-PAGE|EOP imperative-statement-2 ]
     ~~~    ~~~~~~~~~~~ ~~~

   [ INVALID KEY imperative-statement-3 ]
     ~~~~~~~

   [ NOT INVALID KEY imperative-statement-4 ]
     ~~~ ~~~~~~~

[ END-WRITE ]
  ~~~~~~~~~
```

The `WRITE` statement writes a new record to an open file.

1. The reserved words `ADVANCING`, `AT`, `KEY`, `LINE`, `LINES` and `WITH` are optional and may be omitted. The presence or absence of these words has no effect upon the program.
2. The reserved words `END-OF-PAGE` and `EOP` are interchangeable.
3. The *record-name-1* specified on the statement must be defined as an 01-level record subordinate to the File Description (`FD` (see [File/Sort-Description], page 124)) of a file that is currently open for `OUTPUT` (see [File OPEN Modes], page 359), `EXTEND` or `I-O`.
4. The optional `FROM` clause will cause *literal-1* or *identifier-1* to be automatically moved into *record-name-1* prior to writing *record-name-1*'s contents to the appropriate file. If this clause is not specified, it is the programmer's responsibility to populate *record-name-1* with the desired data prior to executing the `WRITE`.
5. The optional `LOCK` options may be used to manually control access to the just-written record by other programs while this program is running. See [Record Locking], page 59, to review the various record locking behaviour.

6. The optional `INVALID KEY` and `NOT INVALID KEY` clauses may be used when writing to relative or indexed files to detect and react to the failure (non-zero file status code) or success (00 file status code), respectively, of the statement. See [File Status Codes], page 107, for additional information.
7. When `WRITE` is used against an `ORGANIZATION LINE SEQUENTIAL` (see [ORGANIZATION LINE SEQUENTIAL], page 111) file, with or without the `LINE ADVANCING` (see [LINE ADVANCING], page 15) option, an end-of-record delimiter character sequence will be written to the file to signify where one record ends and the next record begins. This delimiter sequence will be either of the following:
   - A line-terminator sequence consisting of an ASCII carriage-return/line-feed character sequence (`X'0D0A'`) if you are running a MinGW or native Windows build of GnuCOBOL
   - A line-terminator sequence consisting of an ASCII line-feed character (`X'0A'`) if you are running a Cygwin, Linux, Unix or OSX build of GnuCOBOL
8. The following points pertain to the use (or not) of the `ADVANCING` clause:
   A. Using this clause with any organization other than `ORGANIZATION LINE SEQUENTIAL` will either be rejected outright by the compiler (relative or indexed files) or may introduce unwanted characters into the file (`ORGANIZATION SEQUENTIAL` (see [ORGANIZATION SEQUENTIAL], page 109)).
   B. If no `ADVANCING` clause is specified on a `WRITE` to a line-advancing file, `AFTER ADVANCING 1 LINE` will be assumed; on other than line-advancing files, `BEFORE ADVANCING 1 LINE` will be assumed.
   C. When `BEFORE ADVANCING` is used (or implied), the record is written to the file before the `ADVANCING` action writes line-terminator characters to the file.
   D. If `AFTER ADVANCING` is used (or implied), the `ADVANCING` action writes line-terminator characters to the file and then the record data is written to the file.
   E. The `ADVANCING n LINES` clause will introduce the specified number of line-terminator character sequences into the file either before the written record (`AFTER ADVANCING`) or after the written record (`BEFORE ADVANCING`).
   F. If the `LINAGE` (see [File/Sort-Description], page 124) clause is *absent* from the file's `FD`:
      a. The `ADVANCING PAGE` clause will introduce an ASCII formfeed character into the file either before the written record (`AFTER PAGE`) or after the written record (`BEFORE PAGE`).
      b. Management of areas on the printed page such as top-of page headers, bottom-of-page footers, dealing with "full page" situations and the like are the complete responsibility of the programmer.
   G. If the LINAGE clause is *present* in the file's `FD`:
      a. The `ADVANCING PAGE` clause will introduce the appropriate number of line-terminator character sequences into the file either before the written record (`AFTER ADVANCING`) or after the written record (`BEFORE ADVANCING`) so as to force the printer to automatically advance to a new sheet of paper when the

file prints. No formfeed characters will be generated when `LINAGE` is specified — instead, it is assumed that the printer to which the report will be printed will be loaded with special forms that conform to the specifications defined by the `LINAGE` clause.

b. Management of areas on the printed page such as top-of page headers, bottom-of-page footers, dealing with "full page" situations and the like are now the joint responsibility of the programmer and the GnuCOBOL run-time library, which provides tools such as the `LINAGE-COUNTER` special register (see [Special Registers], page 265) and the `END-OF-PAGE` clause to deal with page formatting issues.

c. The `AT END-OF-PAGE` clause will be triggered, thus executing *imperative-statement-1* (see [Imperative Statement], page 722), if the `WRITE` statement introduces a data line or line-feed character into the file at a line position within the Page Footer area defined by the `LINAGE` clause. The `NOT AT END-OF-PAGE` clause will be triggered (thus executing *imperative-statement-2*) if no end-of-page condition occurred during the `WRITE`.

End of Chapter 7 — PROCEDURE DIVISION

8. Functions

8.1. Intrinsic Functions

GnuCOBOL supports a wide variety of "intrinsic functions" that may be used anywhere in the PROCEDURE DIVISION where a literal is allowed. For example:

```
MOVE FUNCTION LENGTH(Employee-Last-Name) TO Employee-LN-Len
```

Note how the word `FUNCTION` is part of the syntax when you use an intrinsic function. You can use intrinsic functions without having to include the reserved word `FUNCTION` via settings in the `REPOSITORY` (see [REPOSITORY], page 101) paragraph. You may accomplish the same thing by specifying the `-fintrinsics` switch to the GnuCOBOL compiler when you compile your programs.

User-written functions (see [Subprogram Types], page 679) never require the `FUNCTION` keyword when they are executed, because each user-written function a program uses *must* be included in that program's `REPOSITORY` paragraph, which therefore makes the `FUNCTION` keyword optional.

The following intrinsic functions, known to other "dialects" of COBOL, are defined to GnuCOBOL as reserved words but are not otherwise implemented currently. Any attempts to use these functions will result in a compile-time error message. However they are described at the end of this chapter.

```
BOOLEAN-OF-INTEGER
CHAR-NATIONAL
DISPLAY-OF
EXCEPTION-FILE-N
EXCEPTION-LOCATION-N
INTEGER-OF-BOOLEAN
NATIONAL-OF
STANDARD-COMPARE
```

```
Date and Time Formats
~~~~~~~~~~~~~~~~~~~~~
```

For functions `FORMATTED-CURRENT-DATE`, `FORMATTED-DATE`, `FORMATTED-TIME`, and `FORMATTED-DATETIME`, the format literal argument indicates the format of the date or time value that is the result of the function. The result of the function will have the same type as its format literal, which can be alphanumeric, national or UTF-8.

For functions `INTEGER-OF-FORMATTED-DATE`, `SECONDS-FROM-FORMATTED-TIME`, and `TEST-FORMATTED-DATETIME`, the format literal indicates the format of the date or time value specified as the second argument of the function.

The permissible format strings are listed as follows. For a full description of each subfield in the format literals, including a range of permissible values in data associated with the formats, see the Value meanings and limits section.

```
Integer date form:
~~~~~~~~~~~~~~~~~
```

A value in integer date form is a positive integer that represents the number of days since 31 December, 1600 in the Gregorian calendar.

It must be greater than zero and less than or equal to the value of FUNCTION INTEGER-OF-DATE (99991231), which is 3,067,671.

```
Standard date form:
~~~~~~~~~~~~~~~~~~
```

A value in standard date form is an integer of the form YYYYMMDD, calculated using (YYYY * 10,000) + (MM * 100) + DD, where:

YYYY represents the year in the Gregorian calendar. It must be an integer in the range [1601, 9999].

MM represents a month and must be an integer in the range [01, 12].

DD represents a day and must be an integer in the range [01, 31], valid for the specified month and year combination.

```
Julian date form:
~~~~~~~~~~~~~~~~
```

A value in Julian date form is an integer of the form YYYYDDD, calculated using (YYYY * 1000) + DDD, where:

YYYY represents the year in the Gregorian calendar. It must be an integer in the range [1601, 9999].

DDD represents the day of the year. It must be a positive integer in the range [1, 366], valid for the year specified.

```
UTC offset value:
~~~~~~~~~~~~~~~~
```

A UTC offset value is an integer representation of offset from UTC (Coordinated Universal Time) expressed in minutes. The value must be greater than or equal to -1439 and less than or equal to 1439.

Note: The offset value 1439 represents 23 hours 59 minutes, which is one minute less than a day. Standard numeric time form

A value in standard numeric time form is a numeric value representing seconds past midnight. The value must be greater than or equal to zero and less than 86,400

```
Date and time formats:
~~~~~~~~~~~~~~~~~~~~~
```

For functions FORMATTED-CURRENT-DATE, FORMATTED-DATE, FORMATTED-TIME, and FORMATTED-DATETIME, the format literal argument indicates the format of the date or time value that is the result of the function. The result of the function will have the same type as its format literal, which can be alphanumeric, national or UTF-8.

For functions INTEGER-OF-FORMATTED-DATE, SECONDS-FROM-FORMATTED-TIME, and TEST-FORMATTED-DATETIME, the format literal indicates the format of the date or time value specified as the second argument of the function.

The permissible format strings are listed as follows. For a full description of each subfield in the format literals, including a range of permissible values in data associated with the formats, see Value meanings and limits.

```
Date formats            Format literals
~~~~~~~~~~~~            ~~~~~~~~~~~~~~~

Basic calendar date     YYYYMMDD
Extended calendar date  YYYY-MM-DD
Basic ordinal date      YYYYDDD
Extended ordinal date   YYYY-DDD
Basic week date         YYYYWwwD
Extended week date      YYYY-Www-D

Integer-seconds time formats:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Integer-seconds time formats            Format literals

Basic local time                        hhmmss
Extended local time                     hh:mm:ss
Basic Coordinated Universal Time (UTC)  hhmmssZ
Extended UTC time                       hh:mm:ssZ
Basic offset time                       hhmmss+hhmm
Extended offset time                    hh:mm:ss+hh:mm

Fractional-seconds time formats:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Fractional-seconds time formats         Format literals
Basic local time                        hhmmss.ssss
Extended local time                     hh:mm:ss.ssss
```

```
Basic Coordinated Universal Time (UTC)  hhmmss.ssssZ
Extended UTC time                       hh:mm:ss.ssssZ
Basic offset time                       hhmmss.ssss+hhmm
Extended offset time                    hh:mm:ss.ssss+hh:mm
```

Note: The period is used as the decimal separator, and four "s" characters after the period are used for illustrative purposes. The number of "s" characters that might be specified after the decimal separator in these formats might range from 1 to 9.

```
Value meanings and limits:
~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
The permissible date and time formats have the following meanings and limits:

Format    Meaning and limits
YYYY      Year, 1601-9999
MM        Month, 01-12
DD        Day of month, 01-{28|29|30|31} dependent on month sub-field
DDD       Day of year for ordinal date formats, 001-365|366
ww        Week of year, 01-53
D         Day of week, 1-7
W
-
hh        Hours, 00-23
mm        Minutes, 00-59
ss        Seconds, 00-59
.s        Fractional seconds, always prefixed with '.' then 1-9 's'
+|-hh:mm  UTC offset hours (extended times only), the offset can be adjusted
          upward (by a '+' prefix) or downward (by a - prefix). A prefix of 0
          (zero) indicates that an offset of UTC is not available on the system.
Z         UTC time indicator
```

```
Value meanings and limits:
~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
The permissible date and time formats have the following meanings and limits:

Format  Meaning and limits:
~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
YYYY      Year, 1601-9999
MM        Month, 01-12
DD        Day of month, 01-{28|29|30|31} dependent on month sub-field
DDD       Day of year for ordinal date formats, 001-365|366
```

```
ww          Week of year, 01-53
D           Day of week, 1-7
W
-
hh          Hours, 00-23
mm          Minutes, 00-59
ss          Seconds, 00-59
.s          Fractional seconds, always prefixed with '.' then 1-9 's'
+|-hh:mm    UTC offset hours (extended times only), the offset can be adjusted
            upward (by a '+' prefix) or downward (by a - prefix).

            A prefix of 0 (zero) indicates that an offset of UTC is not available
            on the system.
Z                UTC time indicator
```

The supported intrinsic functions are listed in the following sections, along with their syntax and usage notes.

8.1.1. ABS

ABS Function Syntax

```
ABS(number)
~~~
```

This function determines and returns the absolute value of *number* (a numeric literal or data item) supplied as an argument.

Note that `ABSOLUTE-VALUE` has an alias for this function.

8.1.2. ACOS

ACOS Function Syntax

```
ACOS(cosine)
~~~~
```

The `ACOS` function determines and returns the trigonometric arc-cosine, or inverse cosine, of *cosine* value (a numeric literal or data item) supplied as an argument.

The result will be an angle, expressed in radians. You may convert this to an angle measured in degrees, as follows:

```
COMPUTE degrees = ( radians * 180 ) / FUNCTION PI
```

8.1.3. ANNUITY

ANNUITY Function Syntax

```
ANNUITY(interest-rate, number-of-periods)
~~~~~~~
```

This function returns a numeric value approximating the ratio of an annuity paid at *interest-rate* (numeric data item or literal) for each of *number-of-periods* (numeric data items or literals).

interest-rate is the rate of interest paid at each payment. If you only have an annual interest rate and you wish to compute monthly annuity payments, divide the annual interest rate by 12 and use that value for *interest-rate*.

Multiply the result of this function times the desired principal amount to determine the amount of each period's payment.

A note for the financially challenged: an annuity is basically a reverse loan; an accountant would take the result of this function multiplied by -1 times the principal amount to compute a loan payment you are making.

8.1.4. ASIN

ASIN Function Syntax

```
ASIN(sine)
~~~~
```

The `ASIN` function determines and returns the trigonometric arc-sine, or inverse sine, of *sine* value (a numeric literal or data item) supplied as an argument.

The result will be an angle, expressed in radians. You may convert this to an angle measured in degrees, as follows:

```
COMPUTE degrees = ( radians * 180 ) / FUNCTION PI
```

8.1.5. ATAN

ATAN Function Syntax

```
ATAN(tangent)
~~~~
```

Use this function to determine and return the trigonometric arc-tangent, or inverse tangent, of *tangent* value (a numeric literal or data item) supplied as an argument.

The result will be an angle, expressed in radians. You may convert this to an angle measured in degrees, as follows:

```
COMPUTE degrees = ( radians * 180 ) / FUNCTION PI
```

8.1.6. BIT-OF

BIT-OF Function Syntax

```
BIT-OF (argument-1)
~~~~~~
```

`BIT-OF` function returns an alphanumeric character string of '1' and '0' characters, which represents the binary value of each byte in the argument used on input.

1. The function type is alphanumeric.
2. *argument-1* must be a data item, literal, or an intrinsic function result of any data class.

Returned values:

1. An alphanumeric character string consisting of the binary representation of each byte in *argument-1*.
2. The length of the character string returned, in bytes, is eight times the length of *argument-1*, in bytes.

8.1.7. BIT-TO-CHAR

BIT-TO-CHAR Function Syntax

```
BIT-TO-CHAR {argument-1)
~~~~~~~~~~~
```

`BIT-TO-CHAR` function returns a character string that represents a bit pattern supplied on input.

1. The function type is alphanumeric.
2. *argument-1* must be an alphanumeric literal, alphanumeric data item, or alphanumeric group item.
3. *argument-1* must consist only of the characters '0' and '1'.
4. The length of *argument-1* must be a multiple of 8 bytes.

Returned values:

1. A character string consisting of bytes representing the sequence of '0' and '1' characters in *argument-1*.
2. The length of the result string is equal to the length of the input string divided by 8.

8.1.8. BYTE-LENGTH

BYTE-LENGTH Function Syntax

```
BYTE-LENGTH(string)
~~~~~~~~~~~~
```

`BYTE-LENGTH` returns the length — in bytes — of *string* (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal). This intrinsic function is identical to the `LENGTH-AN` (see [LENGTH-AN], page 470) function. Note that the value returned by this function is not necessarily the number of *characters* comprising *string*, but rather the number of actual *bytes* required to store it.

For example, if *string* is encoded using a double-byte character set such as Unicode UTF-16 (where each character is represented by 16 bits of storage, not the 8-bits inherent to character sets like ASCII or EBCDIC), then calling this function with a *string* argument whose `PICTURE` (see [PICTURE], page 194) is `N(4)` would return a value of 8 rather than the value 4.

Contrast this with the `LENGTH` (see [LENGTH], page 469) function.

8.1.9. CHAR

CHAR Function Syntax

```
CHAR(integer)
~~~~
```

This function returns the character in the ordinal position specified by *integer* (a numeric integer literal or data item with a value of 1 or greater) from the `COLLATING SEQUENCE` (see [OBJECT-COMPUTER], page 90) being used by the program.

For example, if the program is using the (default) ASCII character set, CHAR(34) returns the 34th character in the ASCII character set — an exclamation-point ('`!`'). If you are using this function to convert a numeric value to its corresponding ASCII character, you must use an argument value one greater than the numeric value.

If an argument whose value is less than 1 or greater than 256 is specified, the character in the program collating sequence corresponding to a value of all zero bits is returned.

The following code is an alternative approach when you just wish to convert a number to its ASCII equivalent:

```
01  Char-Value.
    05 Numeric-Value        USAGE BINARY-CHAR.
...
    MOVE numeric-character-value TO Numeric-Value
```

The `Char-Value` item now has the corresponding ASCII character value.

8.1.10. COMBINED-DATETIME

COMBINED-DATETIME Function Syntax

```
COMBINED-DATETIME(days, seconds)
~~~~~~~~~~~~~~~~~
```

This function returns a 12-digit numeric result, the first seven digits of which are the integer value of *days* argument (a numeric data item or literal) and the last five of which are the integer value of *seconds* argument (also a numeric data item or literal).

If *days* is less than 1 or greater than 3,067,671, or if *seconds* is less than 1 or greater than 86,400, a value of 0 is returned and a runtime error will result.

days Must be in integer date form. For details, see Integer date form. A value in integer date form is a positive integer that represents a number of days succeeding 31 December 1600, in the Gregorian calendar. It is based on a starting date of Monday, 1 January 1601 and integer date 1 represents Monday, 1 January 1601.

seconds Must be in standard numeric time form. For details, see Standard numeric time form. A value in standard numeric time form is a numeric value representing seconds past midnight.

The returned value is determined by arithmetic expression Days-1 + (Seconds-2/100000). The date occupies the integer part of the returned value and the time is represented in the fractional part of the returned value.

Example Given the integer date form value "143951", which represents the date 15 February 1995, and the standard numeric time form value "18867.812479168304", which represents the time "05:14:27.812479168304", the returned value would be exactly "143951.1886781247".

8.1.11. CONCAT

CONCAT Function Syntax

```
CONCAT | CONCATENATE (argument-1 [, argument-2 ]...)
 ~~~~~~   ~~~~~~~~~~
```

This function concatenates the *argument-1*, *argument-2*, ... (group items, `USAGE DISPLAY` elementary items and/or alphanumeric literals) together into a single string result.

If a numeric literal or `PIC 9` identifier is specified as an argument, decimal points, if any, will be removed and negative signs in `PIC S9` fields or numeric literals will be inserted as defined by the `SIGN IS` (see [SIGN IS], page 213) clause (or absence thereof) of the field. Numeric literals are processed as if `SIGN IS TRAILING SEPARATE` were in effect.

8.1.11. CONCATENATE

CONCATENATE Function Syntax

```
CONCAT | CONCATENATE (argument-1 [, argument-2 ]...)
```

This function concatenates the *string-1*, *string-2*, ... (group items, `USAGE DISPLAY` elementary items and/or alphanumeric literals) together into a single string result.

If a numeric literal or `PIC 9` identifier is specified as an argument, decimal points, if any, will be removed and negative signs in `PIC S9` fields or numeric literals will be inserted as defined by the `SIGN IS` (see [SIGN IS], page 213) clause (or absence thereof) of the field. Numeric literals are processed as if `SIGN IS TRAILING SEPARATE` were in effect.

CONCATENATE is a GnuCOBOL extention BUT also see the ISO standard CONCAT function.

8.1.12. CONTENT-LENGTH

CONTENT-LENGTH Function Syntax

```
CONTENT-LENGTH argument-1
~~~~~~~~~~~~~~
```

Scans for a NUL byte delimiter of the data starting at address in given pointer, and returns the length. The NUL byte is not included in the count. An EC-DATA-PTR-NUL exception is set to exist if the pointer is NUL, and a zero length is returned.

Function CONTENT-LENGTH is a GnuCOBOL extention.

Example:

```
 01  ptr USAGE POINTER.
 01  str  PIC X(4)  VALUE z"abc".

     SET      ptr TO ADDESS OF str.
     DISPLAY  FUNCTION CONTENT-LENGTH (str).

 Will display 3.
```

8.1.13. CONTENT-OF

CONTENT-OF Function Syntax

```
CONTENT-OF pointer-1 { length }
~~~~~~~~~~
```

Takes a pointer and optional length. Returns a character field of the data addressed by the pointer, either up to a NUL byte or to the given length.

The NUL byte is not included in the data when no optional length is given. With an optional count, the character field can hold any content including NUL bytes,

An EC-DATA-PTR-NUL exception is set to exist if the pointer is NUL, and a zero length space is returned.

An EC-SIZE-TRANCATION is set if the resulting field would exceed character field limits and the data is truncated.

Reference modification is allowed on resulting field.

Function CONTENT-OF is a GnuCOBOL extention.

8.1.14. COS

COS Function Syntax

```
COS(angle)
~~~
```

The `COS` function determines and returns the trigonometric cosine of *angle* (a numeric literal or data item) supplied as an argument.

angle is assumed to be a value expressed in radians. If you need to determine the cosine of an angle measured in degrees, you first need to convert that angle to radians as follows:

```
COMPUTE radians = ( degrees * FUNCTION PI) / 180
```

8.1.15. CURRENCY-SYMBOL

CURRENCY-SYMBOL Function Syntax

```
CURRENCY-SYMBOL
~~~~~~~~~~~~~~~
```

The `CURRENCY-SYMBOL` function returns the currency symbol character currently in effect for the locale under which your program is running. On UNIX systems, your locale is established via the `LANG` run-time environment variable (see [Run Time Environment Variables], page 658) environment variable. On Windows, the Control Panel's "Regional and Language Options" define the locale.

Changing the currency symbol via the `SPECIAL-NAMES` (see [SPECIAL-NAMES], page 92) paragraph's `CURRENCY SYMBOL` setting will *not* affect the value returned by this function.

8.1.16. CURRENT-DATE

CURRENT-DATE Function Syntax

```
CURRENT-DATE
~~~~~~~~~~~~
```

Returns the current date and time as the following 21-character structure:

```
01  CURRENT-DATE-AND-TIME.
    05 CDT-Year                PIC 9(4).
    05 CDT-Month               PIC 9(2). *> 01-12
    05 CDT-Day                 PIC 9(2). *> 01-31
    05 CDT-Hour                PIC 9(2). *> 00-23
    05 CDT-Minutes             PIC 9(2). *> 00-59
    05 CDT-Seconds             PIC 9(2). *> 00-59
    05 CDT-Hundredths-Of-Secs  PIC 9(2). *> 00-99
    05 CDT-GMT-Diff-Hours      PIC S9(2)
                               SIGN LEADING SEPARATE.
    05 CDT-GMT-Diff-Minutes    PIC 9(2). *> 00 or 30
```

Since this function has no arguments, no parenthesis should be specified.

8.1.17. DATE-OF-INTEGER

DATE-OF-INTEGER Function Syntax

```
DATE-OF-INTEGER(integer)
~~~~~~~~~~~~~~~
```

This function returns a numeric calendar date in *yyyymmdd* (i.e. Gregorian) format. The date is determined by adding the number of days specified as *integer* (a numeric integer data item or literal) to the date December 31, 1600. For example, `DATE-OF-INTEGER(1)` returns 16010101 while `DATE-OF-INTEGER(150000)` returns 20110908.

A value less than 1 or greater than 3067671 (9999/12/31) will return a result of 0.

8.1.18. DATE-TO-YYYYMMDD

DATE-TO-YYYYMMDD Function Syntax

```
DATE-TO-YYYYMMDD(yymmdd [, yy-cutoff [, yy-execution-time ]])
~~~~~~~~~~~~~~~~
```

You can use this function to convert the six-digit Gregorian date specified as *yymmdd* (a numeric integer data item or literal) to an eight-digit format (*yyyymmdd*).

The optional *yy-cutoff* (a numeric integer data item or literal) argument is the year cutoff used to delineate centuries; if the year component of the date meets or exceeds this cutoff value, the result will be 19yymmdd; if the year component of the date is less than the cutoff value, the result will be 20yymmdd. The default cutoff value if no second argument is given will be 50.

The optional *yy-execution-time* argument (a numeric integer data item or literal) The default execution time value if no third argument is given will be now equivalent to specifying `(FUNCTION NUMVAL (FUNCTION CURRENT-DATE (1:4)))`.

8.1.19. DAY-OF-INTEGER

DAY-OF-INTEGER Function Syntax

```
DAY-OF-INTEGER(integer)
~~~~~~~~~~~~~~
```

This function returns a calendar date in yyyyddd (i.e. Julian) format. The date is determined by adding the number of days specified as integer (a numeric integer data item or literal) to December 31, 1600. For example, `DAY-OF-INTEGER(1)` returns 1601001 while `DAY-OF-INTEGER(250000)` returns 2011251.

A value less than 1 or greater than 3067671 (9999/12/31) will return a result of 0.

8.1.20. DAY-TO-YYYYDDD

DAY-TO-YYYYDDD Function Syntax

```
DAY-TO-YYYYDDD(yyddd [, yy-cutoff [, yy-execution-time ]])
~~~~~~~~~~~~~~
```

You can use this function to convert the five-digit Julian date specified as *yyddd* (a numeric integer data item or literal) to a seven-digit numeric Julian format (yyyyddd).

The optional *yy-cutoff* argument (a numeric integer data item or literal) is the year cutoff used to delineate centuries; if the year component of the date meets or exceeds this cutoff value, the result will be 19yyddd; if the year component of the date is less than the cutoff, the result will be 20yyddd. The default cutoff value if no second argument is given will be 50.

The optional *yy-execution-time* argument (a numeric integer data item or literal) The default execution time value if no third argument is given will be now equivalent to specifying (FUNCTION NUMVAL (FUNCTION CURRENT-DATE (1:4))).

8.1.21. E

E Function Syntax

```
E
~
```

This function returns the mathematical constant *E* (the base of natural logarithms). The maximum precision with which this value may be returned is 2.71828182845904523536028747135266 25.

Since this function has no arguments, no parenthesis should be specified.

8.1.22. EXCEPTION-FILE

EXCEPTION-FILE Function Syntax

```
EXCEPTION-FILE
~~~~~~~~~~~~~~
```

This function returns I/O exception information from the most-recently executed input or output statement. The information is returned as a 34-character string, where the first two characters are the two-digit file status value (see [File Status Codes], page 107) and the remaining 32 are the *file-name-1* specification from the file's `SELECT` (see [SELECT], page 104) statement.

The name returned after the file status information will be returned only if the returned file status value is not 00.

Since this function has no arguments, no parenthesis should be specified.

The documentation of the `CBL_ERROR_PROC` built-in system subroutine (see [CBL_ERROR_PROC], page 568) built-in subroutine illustrates the use of this function.

8.1.23. EXCEPTION-LOCATION

EXCEPTION-LOCATION Function Syntax

```
EXCEPTION-LOCATION
~~~~~~~~~~~~~~~~~~
```

This function returns exception information from the most-recently failing statement. The information is returned to a 1023 character string in one of the following formats, depending on the nature of the failure:

- primary-entry-point-name; paragraph OF section; statement-number
- primary-entry-point-name; section; statement-number
- primary-entry-point-name; paragraph; statement-number
- primary-entry-point-name; statement-number

Since this function has no arguments, no parenthesis should be specified.

The program must be compiled with the `-debug` switch, `-ftraceall` switch or `-g` switch for this function to return any meaningful information.

The documentation of the `CBL_ERROR_PROC` built-in system subroutine (see [CBL_ERROR_PROC], page 568) built-in subroutine illustrates the use of this function.

8.1.24. EXCEPTION-STATEMENT

EXCEPTION-STATEMENT Function Syntax

```
EXCEPTION-STATEMENT
~~~~~~~~~~~~~~~~~~~
```

This function returns the most-recent COBOL statement that generated an exception condition.

Since this function has no arguments, no parenthesis should be specified.

The program must be compiled with the `-debug` switch, `-ftraceall` switch or `-g` switch for this function to return any meaningful information.

The documentation of the `CBL_ERROR_PROC` built-in system subroutine (see [CBL_ERROR_PROC], page 568) built-in subroutine illustrates the use of this function.

8.1.25. EXCEPTION-STATUS

EXCEPTION-STATUS Function Syntax

```
EXCEPTION-STATUS
~~~~~~~~~~~~~~~~
```

This function returns the error type (a text string — see column 2 of the upcoming table for the possible values) from the most-recent COBOL statement that generated an exception condition.

Since this function has no arguments, no parenthesis should be specified.

The documentation of the `CBL_ERROR_PROC` built-in system subroutine (see [CBL_ERROR_PROC], page 568) built-in subroutine illustrates the use of this function.

The following are the error type strings, and their corresponding exception codes and descriptions.

Code	Error Type	Description
0101	EC-ARGUMENT-FUNCTION	Function argument error
0202	EC-BOUND-ODO	OCCURS ... DEPENDING ON data item out of bounds
0204	EC-BOUND-PTR	Data-pointer contains an address that is out of bounds
0205	EC-BOUND-REF-MOD	Reference modifier out of bounds
0207	EC-BOUND-SUBSCRIPT	Subscript out of bounds
0303	EC-DATA-INCOMPATIBLE	Incompatible data exception
0500	EC-I-O	input-output exception
0501	EC-I-O-AT-END	I-O status 1x
0502	EC-I-O-EOP	An end of page condition occurred
0504	EC-I-O-FILE-SHARING	I-O status 6x
0505	EC-I-O-IMP	I-O status 9x
0506	EC-I-O-INVALID-KEY	I-O status 2x

0508	EC-I-O-LOGIC-ERROR	I-O status 4x
0509	EC-I-O-PERMANENT-ERROR	I-O status 3x
050A	EC-I-O-RECORD-OPERATION	I-O status 5x
0601	EC-IMP-ACCEPT	Implementation-defined accept condition
0602	EC-IMP-DISPLAY	Implementation-defined display condition
0A00	EC-OVERFLOW	Overflow condition
0A02	EC-OVERFLOW-STRING	STRING overflow condition
0A03	EC-OVERFLOW-UNSTRING	UNSTRING overflow condition
0B05	EC-PROGRAM-NOT-FOUND	Called program not found
0D03	EC-RANGE-INSPECT-SIZE	Size of replace item in inspect differs
1000	EC-SIZE	Size error exception
1004	EC-SIZE-OVERFLOW	Arithmetic overflow in calculation
1005	EC-SIZE-TRUNCATION	Significant digits truncated in store
1007	EC-SIZE-ZERO-DIVIDE	Division by zero
1202	EC-STORAGE-NOT-ALLOC	The data-pointer specified in a FREE statement does not identify currently allocated storage
1203	EC-STORAGE-NOT-AVAIL	The amount of storage requested by an ALLOCATE statement is not available

8.1.26. EXP

EXP Function Syntax

```
EXP(number)
~~~
```

Computes and returns the value of the mathematical constant *e* raised to the power specified by *number* (a numeric literal or data item).

8.1.27. EXP10

EXP10 Function Syntax

```
EXP10(number)
~~~~~
```

Computes and returns the value of 10 raised to the power specified by *number* (a numeric literal or data item).

8.1.28. FACTORIAL

FACTORIAL Function Syntax

```
FACTORIAL(number)
~~~~~~~~~
```

This function computes and returns the factorial value of *number* (a numeric literal or data item).

8.1.29. FORMATTED-CURRENT-DATE

FORMATTED-CURRENT-DATE Function Syntax

```
FORMATTED-CURRENT-DATE ( argument-1 )
~~~~~~~~~~~~~~~~~~~~~~
```

`FORMATTED-CURRENT-DATE` returns the current date and time provided by the system at run-time, formatted according to date-and-time-format according to the argument type.

FUNCTION FORMATTED-CURRENT-DATE gives you exactly what you asked it to, including up to nanoseconds (8 decimal positions in the seconds) [but the system may only provide miliseconds, especially on older win32].

The function argument must be a national or alphanumeric literal and the content, a combined date and time format.

The returned value is formatted to the same form as *argument-1*.

8.1.30. FORMATTED-DATE

FORMATTED-DATE Function Syntax

```
FORMATTED-DATE ( argument-1, argument-2 )
~~~~~~~~~~~~~~
```

`FORMATTED-DATE` uses a format to convert a date in integer date form to a date in the requested format. The returned value will be in date format.

argument-1 shall be a national or alphanumeric literal.

argument-2 shall be a value in integer date form.

8.1.31. FORMATTED-DATETIME

FORMATTED-DATETIME Function Syntax

```
FORMATTED-DATETIME ( argument-1, argument-2, argument-3, argument-4 )
~~~~~~~~~~~~~~~~~~
```

`FORMATTED-DATETIME` uses a combined time and date form to convert and combine a date in integer form and a numeric time expressed as seconds past midnight in UTC.

argument-1 shall be a national or alphanumeric literal.

argument-2 shall be a value in integer date form.

argument-3 shall be a value in standard numeric time form.

argument-4 is an integer specifying the offset from UTC expressed in minutes. If specified but have a value equal or less than 1439.

Note: The offset value 1439 represents 23 hours 59 minutes which is one minutes less than a day.

argument-4 must not be specified if the time portion in *argument-1* is neither a UTC nor an offset format.

The returned value is a representation of the date contained in *argument-2* combined with the time contained in *argument-3* according to the format in *argument-1*.

If the format in *argument-1* indicates that the returned value is to be expressed in UTC, the time portion of the returned value reflects the adjustment of the value in *argument-3* by the offset in *argument-4*.

If the format in *argument-1* indicates that the time is to be returned as an offset from UTC, the value in *argument-3* is reflected directly in the time portion of the returned value and the offset in *argument-4* is reflected directly in the offset portion of the returned value.

8.1.32. FORMATTED-TIME

FORMATTED-TIME Function Syntax

```
FORMATTED-TIME ( argument-1, argument-2, argument-3 )
~~~~~~~~~~~~~~
```

`FORMATTED-TIME` converts a value representing seconds past midnight formatted time of day with optional offset.

argument-1 shall be a national or alphanumeric literal.

argument-2 shall be a value in integer time form.

argument-3 is an integer specifying the offset from UTC expressed in minutes. If specified but have a value equal or less than 1439.

Note: The offset value 1439 represents 23 hours 59 minutes which is one minutes less than a day.

argument-3 must not be specified if the time portion in *argument-1* is neither a UTC nor an offset format.

Returned value :

Is a representation of the standard numeric time contained in *argument-2* according to the format in *argument-1*.

If the format in *argument-1* indicates that the returned value is to be expressed in UTC, the time portion of the returned value reflects the adjustment of the value in *argument-2* by the offset in *argument-3*.

If the format in *argument-1* indicates that the time is to be returned as an offset from UTC, the value in *argument-2* is reflected directly in the time portion of the returned value and the offset in *argument-3* is reflected directly in the offset portion of the returned value.

8.1.33. FRACTION-PART

FRACTION-PART Function Syntax

```
FRACTION-PART(number)
~~~~~~~~~~~~~
```

This function returns that portion of *number* (a numeric data item or a numeric literal) that occurs to the right of the decimal point. `FRACTION-PART(3.1415)`, for example, returns a value of 0.1415. This function is equivalent to the expression:

number -- FUNCTION INTEGER-PART(*number*)

```
Example:
display "base - " FUNCTION FRACTION-PART(FLOATER).
Gives
base - 000.456789
```

1. When moved to a variable, it MUST have a preceding 'V' in the PICTURE, i.e., PIC v(4).

8.1.34. HEX-OF

HEX-OF Function Syntax

```
HEX-OF {argument-1)
~~~~~~
```

`HEX-OF` function returns an alphanumeric character string consisting of a hexadecimal representation of the argument used on input.

1. The type of the function is alphanumeric.
2. *argument-1* must be a data item, literal, or an intrinsic function result of any data class.

Returned values:

1. An alphanumeric character string consisting of a hexadecimal representation of *argument-1*.
2. The length of the character string returned, in bytes, is twice the length of *argument-1*, in bytes.

8.1.35. HEX-TO-CHAR

HEX-TO-CHAR Function Syntax

```
HEX-TO-CHAR {argument-1)
~~~~~~~~~~~
```

`HEX-TO-CHAR` function returns a character string that represents the hexadecimal digit characters supplied on input.

1. The type of the function is alphanumeric.
2. *argument-1* must be an alphanumeric literal, alphanumeric data item, or alphanumeric group item.
3. *argument-1* must consist only of the characters '0' through '9', 'A' through 'F', and 'a' through 'f'.
4. The length of *argument-1* must be a multiple of 2 bytes.

Returned values:

1. A character string of bytes representing the hexadecimal digit characters of *argument-1*.
2. The length of the result string is equal to the length of the input string divided by 2.

8.1.36. HIGHEST-ALGEBRAIC

HIGHEST-ALGEBRAIC Function Syntax

```
HIGHEST-ALGEBRAIC(numeric-identifier)
~~~~~~~~~~~~~~~~~
```

This function returns the highest (i.e. largest or farthest away from 0 in a positive direction if *numeric-identifier* is signed) value that could possibly be stored in *numeric-identifier*.

8.1.37. INTEGER

INTEGER Function Syntax

```
INTEGER(number)
~~~~~~~
```

The `INTEGER` function returns the greatest integer value that is less than or equal to *number* (a numeric literal or data item).

8.1.38. INTEGER-OF-DATE

INTEGER-OF-DATE Function Syntax

```
INTEGER-OF-DATE(date)
~~~~~~~~~~~~~~~~
```

This function converts *date* (a numeric integer data item or literal) — presumed to be a Gregorian calendar form standard date (YYYYMMDD) — to internal date form (the number of days that have transpired since 1600/12/31).

Once in that form, mathematical operations may be performed against the internal date before it is transformed back into a date using the `DATE-OF-INTEGER` (see [DATE-OF-INTEGER], page 443) or `DAY-OF-INTEGER` (see [DAY-OF-INTEGER], page 445) function.

8.1.39. INTEGER-OF-DAY

INTEGER-OF-DAY Function Syntax

```
INTEGER-OF-DAY(date)
~~~~~~~~~~~~~~~
```

This function converts *date* (a numeric integer data item or literal) — presumed to be a Julian calendar form standard date (YYYYDDD) — to internal date form (the number of days that have transpired since 1600/12/31).

Once in that form, mathematical operations may be performed against the internal date before it is transformed back into a date using the `DATE-OF-INTEGER` (see [DATE-OF-INTEGER], page 443) or `DAY-OF-INTEGER` (see [DAY-OF-INTEGER], page 445) function.

8.1.40. INTEGER-OF-FORMATTED-DATE

INTEGER-OF-FORMATTED-DATE Function Syntax

```
INTEGER-OF-FORMATTED-DATE ( argument-1, argument-2 )
~~~~~~~~~~~~~~~~~~~~~~~~~
```

`INTEGER-OF-FORMATTED-DATE` converts a date that is in specified format to integer date form.

argument-1 shall be a national or alphanumeric literal. The content must be either a date format or a combined date and time format.

argument-2 shall be a data item of the same type as *argument-1*.

If *argument-1* is a date format the content of *argument-2* shall be a valid date in that format.

If *argument-1* is a combined date and time format, the content of *argument-2* shall be a valid combined date and time in same format.

8.1.41. INTEGER-PART

INTEGER-PART Function Syntax

```
INTEGER-PART(number)
~~~~~~~~~~~~
```

Returns the integer portion of *number* (a numeric literal or data item).

8.1.42. LENGTH

LENGTH Function Syntax

```
LENGTH(string)
~~~~~~
```

Returns the length — in characters — of *string* (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal).

The value returned by this function is not the number of *bytes* of storage occupied by string, but rather the number of actual *characters* making up the string. For example, if *string* is encoded using a double-byte character set such as Unicode UTF-16 (where each character is represented by 16 bits of storage, not the 8-bits inherent to character sets like ASCII or EBCDIC), then calling this function with a *string* argument whose `PICTURE is X(4)` would return a value of 4 rather than the value 8 (the actual number of bytes of storage occupied by that item).

Contrast this function with the `BYTE-LENGTH` (see [BYTE-LENGTH], page 433) and `LENGTH-AN` (see [LENGTH-AN], page 470) functions.

8.1.43. LENGTH-AN

LENGTH-AN Function Syntax

```
LENGTH-AN(string)
~~~~~~~~~
```

This function returns the length — in bytes of storage — of *string* (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal).

This intrinsic function is identical to the `BYTE-LENGTH` (see [BYTE-LENGTH], page 433) function.

Note that the value returned by this function is not the number of *characters* making up the *string*, but rather the number of actual *bytes* of storage required to store *string*. For example, if *string* is encoded using a double-byte character set such as Unicode UTF-16 (where each character is represented by 16 bits of storage, not the 8-bits inherent to character sets like ASCII or EBCDIC), then calling this function with a *string* argument whose `PICTURE is N(4)` would return a value of 8 rather than the value 4.

Contrast this with the `LENGTH` (see [LENGTH], page 469) function.

8.1.44. LOCALE-COMPARE

LOCALE-COMPARE Function Syntax

```
LOCALE-COMPARE(argument-1, argument-2 [ , locale ])
~~~~~~~~~~~~~~
```

The `LOCALE-COMPARE` function returns a character indicating the result of comparing *argument-1* and *argument-2* using a culturally-preferred ordering defined by a *locale*.

Either or both of the 1st two arguments may be an alphanumeric literal, a group item or an elementary item appropriate to storing alphabetic or alphanumeric data. If the lengths of the two arguments are unequal, the shorter will be assumed to be padded to the right with spaces.

The two arguments will be compared, character by character, against each other until their relationship to each other can be determined. The comparison is made according to the cultural rules in effect for *locale* name or for the current locale if no *locale* argument is specified. Once that relationship is determined, a one-character alphanumeric value will be returned as follows:

- '<' — If *argument-1* is determined to be less than *argument-2*
- '=' — If the two arguments are equal to each other
- '>' — If *argument-1* is determined to be greater than *argument-2*

See [LOCALE Names], page 93, for a list of typically-available locale names.

8.1.45. LOCALE-DATE

LOCALE-DATE Function Syntax

```
LOCALE-DATE(date [, locale ])
~~~~~~~~~~~
```

Converts the eight-digit Gregorian *date* (a numeric integer data item or literal) from yyyymmdd format to the format appropriate to the current locale. On a Windows system, this will be the "short date" format as set using Control Panel.

You may include an optional second argument to specify the *locale* name (group item or `PIC X` identifier) you'd like to use for date formatting. If used, this second argument *must* be an identifier. Locale names are specified using UNIX-standard names.

8.1.46. LOCALE-TIME

LOCALE-TIME Function Syntax

```
LOCALE-TIME(time [, locale ])
~~~~~~~~~~~
```

Converts the four- (hhmm) or six-digit (hhmmss) *time* (a numeric integer data item or literal) to a format appropriate to the current locale. On a Windows system, this will be the "time" format as set using Control Panel.

You may include an optional *locale* name (a group item or `PIC X` identifier) you'd like to use for time formatting. If used, this second argument *must* be an identifier. Locale names are specified using UNIX-standard names.

8.1.47. LOCALE-TIME-FROM-SECONDS

LOCALE-TIME-FROM-SECONDS Function Syntax

```
LOCALE-TIME-FROM-SECONDS(seconds [, locale ])
~~~~~~~~~~~~~~~~~~~~~~~~
```

Converts the number of *seconds* since midnight (a numeric integer data item or literal) to a format appropriate to the current locale. On a Windows system, this will be the "time" format as set using Control Panel.

You may include an optional *locale* name (a group item or `PIC X` identifier) you'd like to use for time formatting. If used, this second argument *must* be an identifier. Locale names are specified using UNIX-standard names.

See [LOCALE Names], page 93, for a list of typically-available locale names.

8.1.48. LOG

LOG Function Syntax

```
LOG(number)
~~~
```

Computes and returns the natural logarithm (base *e*) of *number* (a numeric literal or data item).

8.1.49. LOG10

LOG10 Function Syntax

```
LOG10(number)
~~~~~
```

Computes and returns the base 10 logarithm of *number* (a numeric literal or data item).

8.1.50. LOWER-CASE

LOWER-CASE Function Syntax

```
LOWER-CASE(string)
~~~~~~~~~~
```

This function returns the value of *string* (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal), converted entirely to lower case.

What constitutes a "letter" (or upper/lower case too, for that manner) may be influenced through the use of a `CHARACTER CLASSIFICATION` (see [OBJECT-COMPUTER], page 90).

8.1.51. LOWEST-ALGEBRAIC

LOWEST-ALGEBRAIC Function Syntax

```
LOWEST-ALGEBRAIC(numeric-identifier)
~~~~~~~~~~~~~~~~
```

This function returns the lowest (i.e. smallest or farthest away from 0 in a negative direction if *numeric-identifier* is signed) value that could possibly be stored in *numeric-identifier*.

8.1.52. MAX

MAX Function Syntax

```
MAX(number-1 [, number-2 ]...)
~~~
```

This function returns the maximum value from the specified list of numbers (each *number-n* may be a numeric data item or a numeric literal).

8.1.53. MEAN

MEAN Function Syntax

```
MEAN(number-1 [, number-2 ]...)
~~~~
```

This function returns the statistical mean value of the specified list of numbers (each *number-n* may be a numeric data item or a numeric literal).

8.1.54. MEDIAN

MEDIAN Function Syntax

```
MEDIAN(number-1 [, number-2 ]...)
~~~~~~
```

This function returns the statistical median value of the specified list of numbers (each *number-n* may be a numeric data item or a numeric literal).

8.1.55. MIDRANGE

MIDRANGE Function Syntax

```
MIDRANGE(number-1 [, number-2 ]...)
 ~~~~~~~~
```

The `MIDRANGE` (middle range) function returns a numeric value that is the arithmetic mean (average) of the values of the minimum and maximum numbers from the supplied list. Each *number-n* may be a numeric data items or a numeric literal.

8.1.56. MIN

MIN Function Syntax

```
MIN(number-1 [, number-2 ]...)
~~~
```

This function returns the minimum value from the specified list of numbers (each *number-n* may be a numeric data item or a numeric literal).

8.1.57. MOD

MOD Function Syntax

```
MOD(value, modulus)
~~~
```

This function returns the value of *value* modulo *modulus* (essentially the remainder from the division of *value* by *modulus*). Both arguments may be numeric data items or numeric literals. Either (or both) may have a non-integer value.

8.1.58. MODULE-CALLER-ID

MODULE-CALLER-ID Function Syntax

```
MODULE-CALLER-ID
~~~~~~~~~~~~~~~~
```

This function returns the null string if it is executed within a main program. When executed with a subprogram, it returns the entry-point name of the program that called the subprogram.

The discussion of the `MODULE-TIME` (see [MODULE-TIME], page 491) function includes a sample program that uses this function.

Since this function has no arguments, no parenthesis should be specified.

8.1.59. MODULE-DATE

MODULE-DATE Function Syntax

```
MODULE-DATE
~~~~~~~~~~~
```

This function Returns the date the GnuCOBOL program that is executing the function was compiled, in the form yyyymmdd.

The discussion of the `MODULE-TIME` (see [MODULE-TIME], page 491) function includes a sample program that uses this function.

Since this function has no arguments, no parenthesis should be specified.

8.1.60. MODULE-FORMATTED-DATE

MODULE-FORMATTED-DATE Function Syntax

```
MODULE-FORMATTED-DATE
~~~~~~~~~~~~~~~~~~~~~
```

This function returns the fully-formatted date and time when the program executing the function was compiled. The exact format of this returned string value may vary depending on the operating system and GnuCOBOL build type.

The discussion of the `MODULE-TIME` (see [MODULE-TIME], page 491) function includes a sample program that uses this function.

Since this function has no arguments, no parenthesis should be specified.

8.1.61. MODULE-ID

MODULE-ID Function Syntax

```
MODULE-ID
 ~~~~~~~~~
```

This function returns the primary entry-point name (i.e. the `PROGRAM-ID` or `FUNCTION-ID` of the program. See [IDENTIFICATION DIVISION], page 85, for information on those clauses.

The discussion of the `MODULE-TIME` (see [MODULE-TIME], page 491) function includes a sample program that uses this function.

Since this function has no arguments, no parenthesis should be specified.

8.1.62. MODULE-PATH

MODULE-PATH Function Syntax

```
MODULE-PATH
~~~~~~~~~~~
```

This function returns the full path to the executable version of this GnuCOBOL program. The filename component of this value will be exactly as typed on the command line, down to the use of upper- and lower-case letters and presence (or absence) of any extension.

The discussion of the `MODULE-TIME` (see [MODULE-TIME], page 491) function includes a sample program that uses this function.

Since this function has no arguments, no parenthesis should be specified.

8.1.63. MODULE-SOURCE

MODULE-SOURCE Function Syntax

```
MODULE-SOURCE
~~~~~~~~~~~~~
```

The filename of the source code of the program (as specified on the `cobc` command when the program was compiled) is returned by this function.

The discussion of the `MODULE-TIME` (see [MODULE-TIME], page 491) function includes a sample program that uses this function.

Since this function has no arguments, no parenthesis should be specified.

8.1.64. MODULE-TIME

MODULE-TIME Function Syntax

```
MODULE-TIME
~~~~~~~~~~~
```

This function returns the time the GnuCOBOL program was compiled, in the form hhmmss.

Since this function has no arguments, no parenthesis should be specified.

The following sample program uses all the MODULE- Functions:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DEMOMODULE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
    FUNCTION ALL INTRINSIC.
PROCEDURE DIVISION.
000-Main.
    DISPLAY "MODULE-CALLER-ID      = [" MODULE-CALLER-ID ']'
    DISPLAY "MODULE-DATE           = [" MODULE-DATE ']'
    DISPLAY "MODULE-FORMATTED-DATE = [" MODULE-FORMATTED-DATE ']'
    DISPLAY "MODULE-ID             = [" MODULE-ID ']'
    DISPLAY "MODULE-PATH           = [" MODULE-PATH ']'
    DISPLAY "MODULE-SOURCE         = [" MODULE-SOURCE ']'
    DISPLAY "MODULE-TIME           = [" MODULE-TIME ']'
    STOP RUN
    .
```

The program produces this output when executed:

```
MODULE-CALLER-ID = []
MODULE-DATE = [20180522]
MODULE-FORMATTED-DATE = [May 22 2018 12:43:14]
MODULE-ID = [DEMOMODULE]
MODULE-PATH = [/home/vince/cobolsrc/ACAS/demomodule]
MODULE-SOURCE = [demomodule.cbl]
MODULE-TIME = [124314]
```

8.1.65. MONETARY-DECIMAL-POINT

MONETARY-DECIMAL-POINT Function Syntax

```
MONETARY-DECIMAL-POINT
~~~~~~~~~~~~~~~~~~~~~~
```

`MONETARY-DECIMAL-POINT` returns the character used to separate the integer portion from the fractional part of a monetary currency value according to the rules currently in effect for the locale under which your program is running.

On UNIX (including OSX, Windows/Cygwin and Windows/MinGW) systems, your locale is established via the `LANG` run-time environment variable (see [Run Time Environment Variables], page 658) environment variable. On Windows, the Control Panel's Regional and Language Options define the locale.

Using the `DECIMAL-POINT IS COMMA` (see [SPECIAL-NAMES], page 92) clause in your program will not affect the value returned by this function.

Since this function has no arguments, no parenthesis should be specified.

8.1.66. MONETARY-THOUSANDS-SEPARATOR

MONETARY-THOUSANDS-SEPARATOR Function Syntax

```
MONETARY-THOUSANDS-SEPARATOR
~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

This function returns the character used to separate the thousands digit groupings of monetary currency values according to the rules currently in effect for the locale under which your program is running.

On UNIX (including OSX, Windows/Cygwin and Windows/MinGW) systems, your locale is established via the `LANG` run-time environment variable (see [Run Time Environment Variables], page 658) environment variable. On Windows, the Control Panel's Regional and Language Options define the locale.

Using the `DECIMAL-POINT IS COMMA` (see [SPECIAL-NAMES], page 92) clause in your program will not affect the value returned by this function.

Since this function has no arguments, no parenthesis should be specified.

8.1.67. NUMERIC-DECIMAL-POINT

NUMERIC-DECIMAL-POINT Function Syntax

```
NUMERIC-DECIMAL-POINT
~~~~~~~~~~~~~~~~~~~~
```

This function returns the character used to separate the integer portion of a non-integer numeric item from the fractional part according to the rules currently in effect for the locale under which your program is running.

On UNIX (including OSX, Windows/Cygwin and Windows/MinGW) systems, your locale is established via the `LANG` run-time environment variable (see [Run Time Environment Variables], page 658) environment variable. On Windows, the Control Panel's Regional and Language Options define the locale.

Using the `DECIMAL-POINT IS COMMA` (see [SPECIAL-NAMES], page 92) clause in your program will not affect the value returned by this function.

Since this function has no arguments, no parenthesis should be specified.

8.1.68. NUMERIC-THOUSANDS-SEPARATOR

NUMERIC-THOUSANDS-SEPARATOR Function Syntax

```
NUMERIC-THOUSANDS-SEPARATOR
~~~~~~~~~~~~~~~~~~~~~~~~~~
```

This function returns the character used to separate the thousands digit groupings of numeric values according to the rules currently in effect for the locale under which your program is running.

On UNIX (including OSX, Windows/Cygwin and Windows/MinGW) systems, your locale is established via the `LANG` run-time environment variable (see [Run Time Environment Variables], page 658) environment variable. On Windows, the Control Panel's Regional and Language Options define the locale.

Using the `DECIMAL-POINT IS COMMA` (see [SPECIAL-NAMES], page 92) clause in your program will not affect the value returned by this function.

Since this function has no arguments, no parenthesis should be specified.

8.1.69. NUMVAL

NUMVAL Function Syntax

```
NUMVAL(string)
~~~~~~
```

The NUMVAL function converts a *string* (a group item, USAGE DISPLAY elementary item or alphanumeric literal) to its corresponding numeric value.

The *string* must have any of the following formats, where '#' represents a sequence of one or more decimal digits:

```
#   -#   +#   #-   #+   #CR   #DB
#.#   -#.#   +#.#   #.#-   #.#+   #.#CR   #.#DB
```

There must be at least one digit character in the string.

Leading and/or trailing spaces are allowed, as are spaces before the first digit.

The character period in *argument-1 string*, represents the decimal separator. The character comma in *argument-1* represents the grouping separator. When the DECIMAL-POINT IS COMMA clause is specified, the character comma shall be used in *argument-1* to represent the decimal separator and the character period shall be used to represent the grouping separator.

Note: Locale-based functionality equivalent to NUMVAL can be obtained by using the NUMVAL-C function with the LOCALE keyword. A currency sign is optional in NUMVAL-C. The locale category LC_MONETARY will be used because there is no sign convention specified in locale category LC_NUMERIC.

Returned values:

The returned value is the numeric value represented by *string*.

If it contains a CR, DB, or the minus sign ('-'), the returned value is negative.

8.1.70. NUMVAL-C

NUMVAL-C Function Syntax

```
NUMVAL-C (string [, symbol                ]
~~~~~~~~
                 [, LOCALE locale-name-1 ] [, ANYCASE ])
```

This function converts a *string* (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal) representing a currency value to its corresponding numeric value.

The currency string if any, and any grouping separators preceding the decimal separator are ignored. Optionally, the currency string, sign convention, grouping separator and the decimal separator permitted in the character string may be specified by locale category `LC-MONETARY`, or the currency string may be specified by *symbol*.

The optional *symbol* character represents the currency symbol (a non-space single-character group item, `USAGE DISPLAY` elementary item or alphanumeric literal) that may be used as the currency character in *string*. Any spaces including leading or trailing are ignored. If no *symbol* is specified, the value that would be returned by the `CURRENCY-SYMBOL` intrinsic function (see [CURRENCY-SYMBOL], page 441) will be used.

If this references the `LOCALE` :

Changing the currency symbol via the `SPECIAL-NAMES` paragraph's `CURRENCY SYMBOL` setting will not affect the value returned by this function.

While `NUMVAL-C` will always use the currency symbol that is specified via the `SPECIAL-NAMES` paragraph's `CURRENCY SYMBOL` (or the system default which is currently always '`$`').

string may have any of the following formats, where '#' represents a sequence of one or more decimal digits and '$' represents the *symbol* character:

```
                  #   -#   +#   #-   #+   #CR   #DB
              #.#   -#.#   +#.#   #.#-   #.#+   #.#CR   #.#DB
                 $#   -$#   +$#   $#-   $#+   $#CR   $#DB
           $#.#   -$#.#   +$#.#   $#.#-   $#.#+   $#.#CR   $#.#DB
```

There must be at least one digit character in the string.

Leading and/or trailing spaces are allowed, as are spaces before and/or after the currency symbol, sign, CR and DB characters.

If the `ANYCASE` keyword is used the matching rules for detecting a currency string in *argument-1* are case-insensitive. If the `ANYCASE` keyword is not specified, the matching rules are case-sensitive.

If neither symbol nor the `LOCALE` keyword is specified, there shall be only one currency string used, either the default currency sign or a currency string specified in the `SPECIAL-NAMES` paragraph.

The returned value is the numeric value represented by string.

When the `LOCALE` keyword is specified, the returned value is negative if string contains a negative sign.

When the `LOCALE` keyword is not specified, the returning value is negative if string contains CR, DB, or a minus sign.

8.1.70B. NUMVAL-C-V2

NUMVAL-C Function Syntax

```
NUMVAL-C (argument-1 [, argument-2          ]
~~~~~~~~
                     [, LOCALE locale-name-1 ] [, ANYCASE ])
```

This function returns the numeric value represented by the character string specified by *argument-1* and defined as alphanumeric.

argument-2, the currency string if any, and any grouping separators preceding the decimal separator are ignored. Optionally, the currency string, sign convention, grouping separator and the decimal separator permitted in the character string may be specified by locale category `LC-MONETARY`, or the currency string may be specified by *argument-2*.

The optional alphanumeric *argument-2* character represents the currency symbol (a non-space and at least one single-character item, that may be used as the currency character in *argument-1*. Any spaces including leading or trailing are ignored. If no *argument-2* is specified, the value that would be returned by the `CURRENCY-SYMBOL` intrinsic function (see [CURRENCY-SYMBOL], page 441) will be used. *argument-2* must not contain any of the digits - through 9, characters '`*`', '`+`', '`-`', '`,`' or '`.`'; or the two consecutive letters `CR` or `DB`, whether upper or lower case or a combination of both.

argument-2 specifies a currency string that may appear in *argument-1*.

If the `ANYCASE` keyword is specified, the matching rules for detecting a currency string in *argument-1* are case-insensitive. If not specified, the matching rules are case-sensitive.

If neither *argument-2* nor the `LOCALE` keyword is specified, there shall be only one currency string used, either the default currency sign or a currency string specified in the `SPECIAL-NAMES` paragraph.

While `NUMVAL-C` will always use the currency symbol that is specified via the `SPECIAL-NAMES` paragraph's `CURRENCY SYMBOL` (or the system default which is currently always '$') *argument-1* shall have any of the following formats, where '#' represents a sequence of one or more decimal digits and '$' represents the *symbol* character:

```
#   -#   +#   #-   #+   #CR   #DB
#.#   -#.#   +#.#   #.#-   #.#+   #.#CR   #.#DB
$#   -$#   +$#   $#-   $#+   $#CR   $#DB
$#.#   -$#.#   +$#.#   $#.#-   $#.#+   $#.#CR   $#.#DB
```

There must be at least one digit character in the string.

Leading and/or trailing spaces are allowed, as are spaces before and/or after the currency symbol, sign, CR and DB characters.

The returned value is the numeric value represented by *argument-1*.

When the `LOCALE` keyword is specified, the returned value is negative if string contains a negative sign and when not specified, the returning value is negative if string contains CR, DB, or a minus sign.

8.1.71. NUMVAL-F

NUMVAL-F Function Syntax

```
NUMVAL-F(char)
~~~~~~~~
```

This function converts a *string* (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal) representing a floating-point value to its corresponding numeric value.

```
#   -#   +#   #E#   -#E#   +#E#
#E+#   -#E+#   +#E+#   #E-#   -#E-#   +#E-#
#.#   -#.#   +#.#   #.#E#   -#.#E#   +#.#E#
#.#E+#   -#.#E+#   +#.#E+#   #.#E-#   -#.#E-#   +#.#E-#
```

There must be at least one digit character both before and after the `E` in the string.

Leading and/or trailing spaces are allowed, as are spaces before and/or after any sign characters.

8.1.72. ORD

ORD Function Syntax

```
ORD(char)
~~~
```

This function returns the ordinal position in the program character set (usually ASCII) corresponding to the 1st character of *char* argument (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal).

For example, assuming the program is using the standard ASCII collating sequence, `ORD('!')` returns 34 because '`!`' is the 34th ASCII character. If you are using this function to convert an ASCII character to its numeric value, you must subtract one from the result.

The following code is an alternative approach when you just wish to convert an ASCII character to its numeric equivalent:

```
01  Char-Value.
    05 Numeric-Value        USAGE BINARY-CHAR.
...
    MOVE "character" TO Char-Value
```

`Numeric-Value` now has the numeric value of `character`.

8.1.73. ORD-MAX

ORD-MAX Function Syntax

```
ORD-MAX(char-1 [, char-2 ]...)
~~~~~~~
```

This function returns the ordinal position in the argument list corresponding to the *char-n* whose 1st character has the highest position in the program collating sequence (usually ASCII).

For example, assuming the program is using the standard ASCII collating sequence, `ORD-MAX('Z', 'z', '!')` returns 2 because the 2nd character in the argument list (the ASCII character '`z`') occurs after '`Z`' and '`!`' in the program collating sequence. Each *char-n* argument may be a group item, `USAGE DISPLAY` elementary item or alphanumeric literal.

8.1.74. ORD-MIN

ORD-MIN Function Syntax

```
ORD-MIN(char-1 [, char-2 ]...)
~~~~~~~
```

This function returns the ordinal position in the argument list corresponding to the *char-n* whose 1st character has the lowest position in the program collating sequence (usually ASCII).

For example, assuming the program is using the standard ASCII collating sequence, `ORD-MIN('Z', 'z', '!')` returns 3 because the 3rd character in the argument list (the ASCII character '`!`') occurs before '`Z`' and '`z`' in the program collating sequence. Each *char-n* argument may be a group item, `USAGE DISPLAY` elementary item or alphanumeric literal.

8.1.75. PI

PI Function Syntax

```
PI
~~
```

This function returns the mathematical constant *PI*. The maximum precision with which this value may be returned is 3.14159265358979323846264338327950 29.

Since this function has no arguments, no parenthesis should be specified.

8.1.76. PRESENT-VALUE

PRESENT-VALUE Function Syntax

```
PRESENT-VALUE(rate, value-1 [, value-2 ])
~~~~~~~~~~~~~~
```

The `PRESENT-VALUE` function returns a value that approximates the present value of a series of future period-end amounts specified by the various *value-n* arguments at a discount rate specified by the *rate* argument.

All arguments are numeric data items and/or numeric literals.

The following equation summarizes how present value is calculated, where N is the number of *value* arguments:

$$presentvalue = \sum_{i=1}^{N} \left(\frac{value_i}{(1 + rate)^i} \right)$$

8.1.77. RANDOM

RANDOM Function Syntax

```
RANDOM[(seed)]
~~~~~~
```

This function returns a pseudo-random non-integer value in the range 0 to 1 (for example, 0.123456789).

The purpose of the optional *seed* argument, is to initialize the chain of pseudo-random numbers that will be returned by the function. Not only will calls to this function using the same *seed* value return the same pseudo-random number, but so will all subsequent executions of the function without a *seed*. This is actually a good thing when you are testing your program because you can rely on always receiving the same sequence of "random" numbers if you always start using the same *seed*.

The *seed* may be any form of literal or data item. If *seed* is numeric, its numeric value will serve as the seed value. If *seed* is alphanumeric, a value for it will be determined as if it were used as an argument to `NUMVAL` (see [NUMVAL], page 496).

Take, for example, the following sample program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DEMORANDOM.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Pseudo-Random-Number        USAGE COMP-1.
PROCEDURE DIVISION.
000-Main.
    MOVE FUNCTION RANDOM(1) TO Pseudo-Random-Number
    DISPLAY Pseudo-Random-Number
    PERFORM 4 TIMES
        MOVE FUNCTION RANDOM    TO Pseudo-Random-Number
        DISPLAY Pseudo-Random-Number
    END-PERFORM
    STOP RUN
    .
```

Every time this program is executed, it will produce the same output, because the same sequence of pseudo-random numbers will be generated:

```
0.41
0.18467
0.63340002
0.26499999
0.19169
```

It is worth mentioning that if the *first* execution of `RANDOM` in your program lacks a *seed* argument, the result will be exactly as if that execution were coded with a *seed* argument value of 1.

Once your program has been thoroughly tested, you'll want different sequences to be generated each time the program runs. One possible way to accomplish this is to use a *seed* that is likely to be different every time the program is executed, as is likely to be the case if the first `MOVE` statement in the previous example were replaced by this:

```
MOVE RANDOM(FUNCTION CURRENT-DATE(1:16))
  TO Pseudo-Random-Number
```

The first 16 characters returned by the `CURRENT-DATE` (see [CURRENT-DATE], page 442) function will be a number in the format *`YYYYMMDDhhmmssnn`*, where *`YYYYMMDD`* is the current calendar date and *`hhmmssnn`* is the current time of day to the one one-hundredth of a second. Since two different executions of the program will never get identical `CURRENT-DATE` values (unless they are executed in extremely close time frames to one another), using those first sixteen characters as the `RANDOM` seed will guarantee that receiving a duplicate sequence of pseudo-random numbers in two different executions of the program will be *highly* unlikely.

8.1.78. RANGE

RANGE Function Syntax

```
RANGE(number-1 [, number-2 ]...)
~~~~~
```

The `RANGE` function returns a value that is equal to the value of the maximum *number-n* in the argument list minus the value of the minimum *number-n* argument.

All *number-n* arguments are numeric data items and/or numeric literals.

8.1.79. REM

REM Function Syntax

```
REM(number,divisor)
~~~
```

This function returns a numeric value that is the remainder of *number* divided by *divisor*. Both arguments must be numeric data items or numeric literals.

8.1.80. REVERSE

REVERSE Function Syntax

```
REVERSE(string)
~~~~~~~
```

This function returns the byte-by-byte reversed value of *string* (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal).

8.1.81. SECONDS-FROM-FORMATTED-TIME

SECONDS-FROM-FORMATTED-TIME Function Syntax

```
SECONDS-FROM-FORMATTED-TIME(format,time)
~~~~~~~~~~~~~~~~~~~~~~~~~~
```

This function decodes the string *time* — whose value represents a formatted time — and returns the total number of seconds that string represents.

The *time* string must contain hours, minutes and seconds. The time argument may be specified as a group item, `USAGE DISPLAY` elementary item or an alphanumeric literal.

The *format* argument is a string (a group item, `USAGE DISPLAY` elementary item or an alphanumeric literal) documenting the format of *time* using ***hh***, ***mm*** and ***ss*** to denote where the respective time information can be found. Any other characters found in *format* represent character positions that will be ignored. For example, a format of `hhmmss` indicates that *time* will be treated as a six-digit string value where the first two characters are the number of hours, the next two represent minutes and the last two represent seconds. A *format* of `hh:mm:ss`, however, describes *time* as an eight-character string where characters 3 and 6 will be ignored.

8.1.82. SECONDS-PAST-MIDNIGHT

SECONDS-PAST-MIDNIGHT Function Syntax

```
SECONDS-PAST-MIDNIGHT
~~~~~~~~~~~~~~~~~~~~
```

This function returns the current time of day expressed as the total number of elapsed seconds since midnight.

Since this function has no arguments, no parenthesis should be specified.

8.1.83. SIGN

SIGN Function Syntax

```
SIGN(number)
~~~~
```

The `SIGN` function returns a -1 if the value of *number* (a numeric literal or numeric data item) is negative, a zero if the value of *number* is exactly zero and a 1 if the value of *number* if greater than 0.

8.1.84. SIN

SIN Function Syntax

```
SIN(angle)
~~~
```

This function determines and returns the trigonometric sine of *angle* (a numeric literal or numeric data item).

The *angle* is assumed to be a value expressed in radians. If you need to determine the sine of an angle measured in degrees, you first need to convert that angle to radians as follows:

```
COMPUTE radians = ( degrees * FUNCTION PI) / 180
```

8.1.85. SQRT

SQRT Function Syntax

```
SQRT(number)
~~~~
```

The SQRT function returns a numeric value that approximates the square root of *number* (a numeric data item or numeric literal with a non-negative value).

The following two statements produce identical results:

```
01  Result           PIC 9(4).9(10).
...
    MOVE FUNCTION SQRT(15) TO Result
    COMPUTE Result = 15 ^ 0.5
```

8.1.86. STANDARD-DEVIATION

STANDARD-DEVIATION Function Syntax

```
STANDARD-DEVIATION(number-1 [, number-2 ]...)
~~~~~~~~~~~~~~~~~~
```

This function returns the statistical standard deviation of the list of *number-n* arguments (numeric data items or numeric literals).

8.1.87. STORED-CHAR-LENGTH

STORED-CHAR-LENGTH Function Syntax

```
STORED-CHAR-LENGTH(string)
~~~~~~~~~~~~~~~~~~
```

Returns the length — in bytes — of the specified `string` (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal), minus the total number of trailing spaces, if any.

8.1.88. SUBSTITUTE

SUBSTITUTE Function Syntax

```
SUBSTITUTE(string, from-1, to-1 [, from-n, to-n ]...)
~~~~~~~~~~
```

This function parses *string*, replacing all occurrences of *from-n* strings with the corresponding *to-n* strings.

The *from-n* strings must match sequences in *string* exactly with regard to value and case.

A *from-n* string does not have to be the same length as its corresponding *to-n* string.

All arguments are group items, `USAGE DISPLAY` elementary items or alphanumeric literals.

A null *to-n* string will be treated as a single space.

When using Variables in place of *string* attention to NOT wanting Leading or trailing spaces usage of function TRIM needs to be utilised as failure to do so will result in variables treated with any unwanted spaces leading and/or trailing, i.e.,

```
move     function SUBSTITUTE-CASE (WS-Dest-File-Path,
                    function TRIM (WS-Inbound-Path),
                    function TRIM (WS-Desc-Path))
                              to WS-Dest-File-Path
```

8.1.89. SUBSTITUTE-CASE

SUBSTITUTE-CASE Function Syntax

```
SUBSTITUTE-CASE(string, from-1, to-1 [, from-n, to-n ]...)
~~~~~~~~~~~~~~~
```

The `SUBSTITUTE-CASE` function operates the same as the `SUBSTITUTE` (see [SUBSTITUTE], page 518) function, except that *from-n* string matching is performed without regard to case.

All arguments are group items, `USAGE DISPLAY` elementary items or alphanumeric literals.

When using Variables in place of *string* attention to NOT wanting Leading or trailing spaces usage of function TRIM needs to be utilised as failure to do so will result in variables treated with any unwanted spaces leading and/or trailing, i.e.,

```
move     function SUBSTITUTE-CASE (WS-Dest-File-Path,
                    function TRIM (WS-Inbound-Path),
                    function TRIM (WS-Desc-Path))
                              to WS-Dest-File-Path
```

8.1.90. SUM

SUM Function Syntax

```
SUM(number-1 [, number-2 ]...)
~~~
```

The `SUM` function returns a value that is the sum of *number-n* arguments (these may be numeric data items or numeric literals).

8.1.91. TAN

TAN Function Syntax

```
TAN(angle)
~~~
```

This function determines and returns the trigonometric tangent of *angle* (a numeric literal or numeric data item).

The *angle* is assumed to be a value expressed in radians. If you need to determine the tangent of an angle measured in degrees, you first need to convert that angle to radians as follows:

```
COMPUTE radians = ( degrees * FUNCTION PI) / 180
```

8.1.92. TEST-DATE-YYYYMMDD

TEST-DATE-YYYYMMDD Function Syntax

```
TEST-DATE-YYYYMMDD(date)
~~~~~~~~~~~~~~~~~~
```

This function determines if the supplied *date* argument (a numeric integer data item or literal) is a valid date.

A valid date is one of the form yyyymmdd in the range 1601/01/01 to 9999/12/31, with no more than the expected maximum number of days in the month, accounting for leap year.

If the *date* is valid, a 0 value is returned. If it isn't, a value of 1, 2 or 3 is returned signalling the problem lies with the year, month or day, respectively.

8.1.93. TEST-DAY-YYYYDDD

TEST-DAY-YYYYDDD Function Syntax

```
TEST-DATE-YYYYDDD(date)
~~~~~~~~~~~~~~~~~
```

This function determines if the supplied *date* (a numeric integer data item or literal) is a valid date.

A valid date is one of the form `yyyyddd` in the range 1601001 to 9999365. Leap year is accounted for in determining the maximum number of days in a year.

If the date is valid, a 0 value is returned. If it isn't, a value of 1 or 2 is returned signalling the problem lies with the year or day, respectively.

8.1.94. TEST-FORMATTED-DATETIME

TEST-FORMATTED-DATETIME Function Syntax

```
TEST-FORMATTED-DATETIME ( argument-1, argument-2 )
~~~~~~~~~~~~~~~~~~~~~~~
```

`TEST-FORMATTED-DATETIME` tests whether a date literal representing a date, a time or a combined date and time is valid according to the specified format.

argument-1 must a literal of type alphanumeric, UTF-8 or national, that contains a date, time or combined data time format. See Date and Time formats for details.

argument-2 must be a data item of the same type as *argument-1*.

Returned value:

If no format or range problems occur during evaluation of *argument-2* according to the format in *argument-1*, the returned value is zero. Otherwise the returned value is the ordinal character position at which the first error in *argument-2* was detected.

Example

Using the following arguments, it will generates a return value of 5, as the fifth character of argument-2 ("4") contains an incorrect value for the first digit of the month representation.

FUNCTION TEST-FORMATTED-DATETIME(YYYYMMDD", "20124523")

8.1.95. TEST-NUMVAL

TEST-NUMVAL Function Syntax

```
TEST-NUMVAL(string)
~~~~~~~~~~~~
```

The `TEST-NUMVAL` function evaluates *string* (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal) for being appropriate for use as the *string* argument to a `NUMVAL` (see [NUMVAL], page 496) function, returning to a integer a zero value if it is appropriate otherwise if one or more characters are in error, the position of the first character in error or the length of the field plus one for other cases such as all spaces.

Note that these errors include but are not limited to: argument (*string*) is zero length, contains only spaces or contains valid characters but is incomplete, such as the string '`+.`'.

8.1.96. TEST-NUMVAL-C

TEST-NUMVAL-C Function Syntax

```
TEST-NUMVAL-C(string[,symbol])
~~~~~~~~~~~~~
```

This function evaluates *string* (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal) for being appropriate for use as the *string* argument to a `NUMVAL-C` (see [NUMVAL-C], page 497) function, returning to a integer a zero value if it is appropriate otherwise if one or more characters are in error, the position of the first character in error or the length of the field plus one for other cases such as all spaces.

Note that these errors include but are not limited to: argument (*string*) is zero length, contains only spaces or contains valid characters but is incomplete, such as the string '`+.`'.

The optional *symbol* argument serves the same function — and has the same default and possible values — as the corresponding argument of the `NUMVAL-C` function.

8.1.97. TEST-NUMVAL-F

TEST-NUMVAL-F Function Syntax

```
TEST-NUMVAL-F(string)
~~~~~~~~~~~~~
```

This function evaluates *string* (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal) for being appropriate for use as the *string* argument to a `NUMVAL-F` (see [NUMVAL-F], page 500) function, returning to a integer a zero value if it is appropriate otherwise if one or more characters are in error, the position of the first character in error or the length of the field plus one for other cases such as all spaces.

Note that these errors include but are not limited to: argument (string) is zero length, contains only spaces or contains valid characters but is incomplete, such as the string '`+.`'.

8.1.98. TRIM

TRIM Function Syntax

```
TRIM(string [, LEADING|TRAILING ])
~~~~           ~~~~~~~ ~~~~~~~~
```

This function removes `LEADING` or `TRAILING` spaces from *string* (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal).

The second argument is specified as a keyword, not a quoted string or identifier. If no second argument is specified, *both* leading and trailing spaces will be removed. The case (upper, lower or mixed) of this argument is irrelevant.

8.1.99. UPPER-CASE

UPPER-CASE Function Syntax

```
UPPER-CASE(string)
~~~~~~~~~~
```

This function returns the value of *string* (a group item, `USAGE DISPLAY` elementary item or alphanumeric literal), converted entirely to upper case.

What constitutes a "letter" (or upper/lower case too, for that manner) may be influenced through the use of a `CHARACTER CLASSIFICATION` (see [OBJECT-COMPUTER], page 90).

8.1.100. VARIANCE

VARIANCE Function Syntax

```
VARIANCE(number-1 [, number-2 ]...)
~~~~~~~~
```

This function returns the statistical variance of the specified list of *number-n* arguments (these may be numeric data items or numeric literals).

8.1.101. WHEN-COMPILED

WHEN-COMPILED Function Syntax

```
WHEN-COMPILED
~~~~~~~~~~~~~
```

The `WHEN-COMPILED` intrinsic function, not to be confused with the `WHEN-COMPILED` (see [Special Registers], page 265) special register, returns the date and time the program was compiled, in ASCII.

Since this function has no arguments, no parenthesis should be specified.

Unlike the `WHEN-COMPILED` special register, which has an ASCII value of the compilation date/time in the format `mm/dd/yyhh.mm.ss`, the `WHEN-COMPILED` intrinsic function returns the compilation date/time as an ASCII string in the format `yyyymmddhhmmssnnooooo`, where `yyyymmdd` is the date, `hhmmss` is the time, `nn` is the hundredths of a second component of the compilation time, if available (or `00` if it isn't) and `ooooo` is the time zone offset from GMT.

If the `-fintrinsics=WHEN-COMPILED` switch or `-fintrinsics=ALL` switch is specified to the compiler or the `REPOSITORY` (see [REPOSITORY], page 101) paragraph specifies either `FUNCTION WHEN-COMPILED INTRINSIC` or `FUNCTION ALL INTRINSIC`, then references to `WHEN-COMPILED` (without a leading `FUNCTION` keyword will always reference this intrinsic function and there will be no way to access the `WHEN-COMPILED` special register.

8.1.102. YEAR-TO-YYYY

YEAR-TO-YYYY Function Syntax

```
YEAR-TO-YYYY(yy [, yy-cutoff [, yy-execution-time ]])
~~~~~~~~~~~~
```

`YEAR-TO-YYYY` converts *yy* — a two-digit year — to a four-digit format (`yyyy`).

The optional *yy-cutoff* argument is the year cutoff used to delineate centuries; if *yy* meets or exceeds this cutoff value, the result will be 19yy; if *yy* is less than the cutoff, the result will be 20yy. The default cutoff value if no second argument is given will be 50.

The optional *yy-execution-time* argument (a numeric integer data item or literal) The default execution time value if no third argument is given will be now equivalent to specifying (FUNCTION NUMVAL (FUNCTION CURRENT-DATE (1:4))).

All arguments must be numeric data items or numeric literals.

8.1.103. BOOLEAN-OF-INTEGER

BOOLEAN-OF-INTEGER Function Syntax

```
BOOLEAN-OF-INTEGER(argument-1 argument-2)
~~~~~~~~~~~~~~~~~~
```

This option is not yet implemented.

The included file NEWS will indicate when it is.

`BOOLEAN-OF-INTEGER` returns a boolean item of usage bit representing the binary value of *argument-1*. *argument-2* specifies the length of the boolean data item that is returned.

argument-1 must be a positive integer.

argument-2 must be a positive non-zero integer

Returned value is a boolean item of usage bit that has the same bit configuration as the binary representation of the value of *argument-1*, where the rightmost boolean position is the low-order binary digit. The boolean value is zero-filled or truncated on the left, if necessary, in order to return a boolean item whose length is specified by *argument-2* in therms of boolean positions.

8.1.104. CHAR-NATIONAL

CHAR-NATIONAL Function Syntax

```
CHAR-NATIONAL(argument-1)
~~~~~~~~~~~~~
```

This option is not yet implemented.

The included file NEWS will indicate when it is.

`CHAR-NATIONAL` returns a one character value that is a character in the national program collating sequence having the ordinal position equal to the value of the argument.

argument-1 must be a integer and greater than zero and less than or equal to the number of positions in the national program collating sequence.

8.1.105. DISPLAY-OF

DISPLAY-OF Function Syntax

```
DISPLAY-OF(argument-1 [ argument-2] )
~~~~~~~~~~
```

This option is not yet implemented.

The included file NEWS will indicate when it is.

`DISPLAY-OF` returns a character string containing the alphabetic coded character set representation of the national characters in the argument.

argument-1 must be of class national.

argument-2 must be a of class alphabetic or alphanumeric and is one character position in length. It specifies an alphanumeric substitution character for use in conversion of national characters for which there is no corresponding alphanumeric character.

A character string is returned with each national character of *argument-1* converted to its corresponding alphanumeric character representation, if any.

If *argument-2* is specified, the alphanumeric substitution character is returned for each national character in *argument-1* that has no corresponding alphanumeric character representation.

If *argument-2* is un-specified, and *argument-1* contains a national character for which there is no corresponding alphanumeric character representation, an substitution character is used as the corresponding alphanumeric character and the EC-DATA-CONVERSION exception condition is set.

The length of the returned value is the number of character positions of usage display required to hold the converted argument and depends on the number of characters contained in *argument-1*.

8.1.106. EXCEPTION-FILE-N

EXCEPTION-FILE-N Function Syntax

```
EXCEPTION-FILE-N
~~~~~~~~~~~~~~~~
```

This option is not yet implemented.

The included file NEWS will indicate when it is.

`EXCEPTION-FILE-N` returns a national character string that is the I/O status value and file-name of the file connector, if any, associated with the last exception status.

The value returned has a length that is based on its contents and the concents are as follows:

If the last exception status is not an `EC-I-O` exception condition, the returned value is two national zeros.

The returned value is two national spaces when the last exception status indicates an `EC-I-O` exception condition that originates from one of the following statements:

- a `RAISE` statement.
- an `EXIT` or a `GOBACK` statement with a `RAISING` phrase that specifies an `EC-I-O` exception-name.

Otherwise the returned value is a character string that is as long as is needed to contain the I-O status value and the filename. The first two characters are the I-O status value in national characters. The succeeding characters contain the file-name exactly as specified in the `SELECT` clause converted at runtime to the runtime national character set.

The documentation of the `CBL_ERROR_PROC` built-in system subroutine (see [CBL_ERROR_PROC], page 568) built-in subroutine illustrates the use of this function.

8.1.107. EXCEPTION-LOCATION-N

EXCEPTION-LOCATION-N Function Syntax

```
EXCEPTION-LOCATION-N
~~~~~~~~~~~~~~~~~~~~
```

This option is not yet implemented.

The included file NEWS will indicate when it is.

`EXCEPTION-LOCATION-N` returns an national character string containing exception information from the most-recently failing statement. The information is returned to a 1023 character string in one of the following formats, depending on the nature of the failure:

- primary-entry-point-name; paragraph OF section; statement-number
- primary-entry-point-name; section; statement-number
- primary-entry-point-name; paragraph; statement-number
- primary-entry-point-name; statement-number

Since this function has no arguments, no parenthesis should be specified.

The program must be compiled with the `-debug` switch, `-ftraceall` switch or `-g` switch for this function to return any meaningful information.

The documentation of the `CBL_ERROR_PROC` built-in system subroutine (see [CBL_ERROR_PROC], page 568) built-in subroutine illustrates the use of this function.

8.1.108. INTEGER-OF-BOOLEAN

INTEGER-OF-BOOLEAN Function Syntax

```
INTEGER-OF-BOOLEAN(argument-1)
~~~~~~~~~~~~~~~~~~
```

This option is not yet implemented.

The included file NEWS will indicate when it is.

`INTEGER-OF-BOOLEAN` returns the numeric value of the boolean string in *argument-1* which is class boolean.

Returned value as *argument-1* is assigned to a temporary boolean data item of usage bit described with the same number of boolean positions.

The unsigned binary value represented by the same bit configuration as the bit configuration of that temporary boolean data item is determined.

Note: Binary representation is a mathematical concept. It is not required that this representation be the same as a COBOL representation.

8.1.109. NATIONAL-OF

NATIONAL-OF Function Syntax

```
NATIONAL-OF(argument-1 [argument-2] )
~~~~~~~~~~~
```

This option is not yet implemented.

The included file NEWS will indicate when it is.

`NATIONAL-OF` returns a character string containing the national character representation of the characters in the argument which must be of class boolean.

A character string is returned with each alphanumeric character in *argument-1* converted to its corresponding national coded character set representation.

If *argument-2* is specified, each character in *argument-1* that has no corresponding national representation is converted to the substitution character specified by *argument-2*.

If *argument-2* is unspecified and *argument-1* contains an alphanumeric character for which there is no corresponding national character representation, a substitution character is used as the corresponding national character and the `EC-DATA-CONVERSION` exception condition is set to exist.

The length of the returned value is the number of character positions of usage national required to hold the converted argument and depends on the number of characters contained in *argument-1*.

8.1.110. STANDARD-COMPARE

STANDARD-COMPARE Function Syntax

```
STANDARD-COMPARE(argument-1 argument-2 [ordering-name-1] [argument-4] )
~~~~~~~~~~~~~~~~
```

This option is not yet implemented.

The included file NEWS will indicate when it is.

`STANDARD-COMPARE` returns a character indicating the result of comparing *argument-1* as a alphanumeric and *argument-2* using a cultural ordering table.

1. *argument-1* shall be of class alphabetic, alphanumeric, or national.
2. *argument-2* shall be of class alphabetic, alphanumeric, or national.
3. *argument-1* and *argument-2* may be of different classes.
4. Neither *argument-1* nor *argument-2* shall be a zero-length literal.
5. *ordering-name-1*, if specified, shall be associated with a cultural ordering table in the `ORDER TABLE` clause of the `SPECIAL-NAMES` paragraph. *ordering-name-1* identifies the ordering table to be used for the comparison. If *ordering-name-1* is not specified, the default ordering table 'ISO14651_2010_TABLE1' described in Appendix A of ISO/IEC 14651:2011 shall be used.
6. *argument-4*, if specified, shall be a positive nonzero integer.

Returned values:

1. If *argument-4* is unspecified, the highest level defined in the ordering table is used for the comparison.
2. If the cultural ordering table is not available on the processor, or the specified ordering level is not available, or the level number specified by *argument-4* is not defined in the ordering table, the `EC-ORDER-NOT-SUPPORTED` exception condition is set.
3. If the arguments are of different classes, and one is national, the other argument is converted to class national for purposes of comparison.
4. For purposes of comparison, trailing spaces are truncated from the operands except that an operand consisting of all spaces is truncated to a single space.
5. *argument-1* and *argument-2* are compared in accordance with the ordering table and ordering level being used.

   Note: This comparison is culturally sensitive and the default ordering table is acceptable for most cultures. It is not necessarily a character-by-character comparison and not necessarily a case-sensitive comparison. In order to use this function, users should understand the types of comparisons specified by ISO/IEC 14651:2D11 and the ordering tables in use for their installation.

6. The returned value is:

   '=' the arguments compare equal,

   '-=.:' *argument-1* is less than *argument-2*,

   ':>' *argument-1* is greater than *argument-2*.

7. The length of the returned value is 1.

8.2. Built-In System Subroutines

There are a number of built-in system subroutines included with GnuCOBOL.

Generally, these routines are intended to match those available in Micro Focus COBOL, ACUCOBOL and directly for GnuCOBOL.

It is recommended to change the CBL_OC routines to CBL_GC for forward compatibility as at some point they will be removed as they are a hangover from Open Cobol.

Prefix explanation:

`C$` –> `ACU`

`CBL_` –> `MF`

`CBL_GC_` (For backwards compatibility some routines are also available as `CBL_OC_` as well): but these wonderful extensions are *only* available with GnuCOBOL.

These routines, all executed via their *upper-case names* via the `CALL` statement (see [CALL], page 293), are capable of performing the following functions:

- Changing the current directory
- Copying files
- Creating a directory
- Creating, Opening, Closing, Reading and Writing byte-stream files
- Deleting directories (folders)
- Deleting files
- Determining how many arguments were passed to a subroutine
- Getting file information (size and last-modification date/time)
- Getting the length (in bytes) of an argument passed to a subroutine
- Justifying a field left-, right- or center-aligned
- Moving files (a destructive "copy")
- Putting the program "to sleep", specifying the sleep time in seconds
- Putting the program "to sleep", specifying the sleep time in nanoseconds; *Caveat*: although you'll express the time in nanoseconds, Windows systems will only be able to sleep at a millisecond granularity
- Retrieving information about the currently-executing program
- Submitting a command to the shell environment appropriate for the version of GnuCOBOL you are using for execution

Early versions of Micro Focus COBOL allowed programmers to access various runtime library routines by using a single two-digit hexadecimal number as the entry-point name. These were known as call-by-number routines. Over time, Micro Focus COBOL evolved, replacing most of the call-by-number routines with ones accessible using a more conventional call-by-name technique.

Most of the call-by-number routines have evolved into even more powerful call-by-name routines, many of which are supported by GnuCOBOL.

Some of the original call-by-number routines never evolved call-by-name equivalents; GnuCOBOL supports some of these routines.

The following sections describe the various built-in subroutines. *All subroutine arguments are mandatory except where explicitly noted to the contrary.* Any subroutine returning a value to the `RETURN-CODE` special register (see [Special Registers], page 265) could utilize the `RETURNING` clause on the `CALL` statement to return the result back to the full-word binary data item of your choice.

8.2.1. C$CALLEDBY

C$CALLEDBY Built-In Subroutine Syntax

```
CALL "C$CALLEDBY" USING prog-name-area
~~~~              ~~~~~
```

This routine returns the name of the program that called the currently-executing program. The program name will be returned, left-justified and space filled, in *prog-name-area* argument, which should be a `PIC X` elementary item or a group item. If *prog-name-area* is too small to receive the entire program name, the program name value will be truncated (on the right) to fit.

The `RETURN-CODE` special register (see [Special Registers], page 265) will be set to one of the following values:

-1 An error occurred. The *prog-name-area* contents will be unchanged.

0 The program calling `C$CALLEDBY` was not called by any other program (in other words, it is a main program). The *prog-name-area* contents will be set entirely to spaces.

1 The program calling `C$CALLEDBY` was indeed called by another program, and that program's name has been saved in *prog-name-area*.

8.2.2. C$CHDIR

C$CHDIR Built-In Subroutine Syntax

```
CALL "C$CHDIR" USING directory-path, result
~~~~           ~~~~~
```

This routine makes *directory-path* (an alphanumeric literal or identifier) the current directory.

The return code of the operation is returned both in the *result* argument (any non-edited numeric identifier) as well as in the `RETURN-CODE` special register (see [Special Registers], page 265). The return code of the operation will be either 0=Success or 128=failure.

The directory change remains in effect until the program terminates (in which the original current directory at the time the program was started will be automatically restored) or until another `C$CHDIR` or a `CBL_CHANGE_DIR` built-in system subroutine (see [CBL_CHANGE_DIR], page 559) is executed.

8.2.3. C$COPY

C$COPY Built-In Subroutine Syntax

```
CALL "C$COPY" USING src-file-path, dest-file-path, 0
~~~~          ~~~~~
```

Use this subroutine to copy file *src-file-path* to *dest-file-path* as if it were done via the `cp` (Unix/OSX) or `COPY` (Windows) command.

Both file path arguments may be alphanumeric literals or identifiers.

The third argument is required, but is unused.

If the attempt to copy the file fails (for example, it or the destination directory doesn't exist), the `RETURN-CODE` special register (see [Special Registers], page 265) will be set to 128; on successful completion it will be set to 0.

8.2.4. C$DELETE

C$DELETE Built-In Subroutine Syntax

```
CALL "C$DELETE" USING file-path, 0
~~~~            ~~~~~
```

This routine deletes the file specified by the *file-path* argument (an alphanumeric literal or identifier) just as if that were done using the `rm` (Unix/OSX) or `ERASE` (Windows) command.

The second argument is required, but is unused.

If the attempt to delete the file fails (for example, it doesn't exist), the `RETURN-CODE` special register (see [Special Registers], page 265) will be set to 128; on successful completion it will be set to 0.

8.2.5. C$FILEINFO

C$FILEINFO Built-In Subroutine Syntax

```
CALL "C$FILEINFO" USING file-path, file-info
~~~~              ~~~~~
```

With this routine you may retrieve the size of the file specified as the *file-path* argument (an alphanumeric literal or identifier) and the date/time that file was last modified. File size information may not be available in the particular GnuCOBOL build / Operating System combination you are using and may therefore always be returned as zero. The information is returned to the *file-info* argument, which is defined as the following 16-byte area:

```
01  File-Info.
    05 File-Size-In-Bytes  PIC 9(18) COMP.
    05 Mod-YYYYMMDD        PIC 9(8)  COMP. *> Modification Date
    05 Mod-HHMMSS00        PIC 9(8)  COMP. *> Modification Time
```

The last two decimal digits in the modification time will always be 00.

If the subroutine is successful, a value of 0 will be returned in the `RETURN-CODE` special register (see [Special Registers], page 265). Failure to retrieve the needed statistics on the file will cause a `RETURN-CODE` special register value of 35 to be passed back. Supplying less than two arguments will generate a 128 `RETURN-CODE` special register value.

8.2.6. C$GETPID

C$GETPID Built-In Subroutine Syntax

```
CALL "C$GETPID"
~~~~
```

Use this subroutine to return the PID (process ID) of the executing GnuCOBOL program. The PID value is returned into the `RETURN-CODE` special register (see [Special Registers], page 265).

There are no arguments to this routine.

8.2.7. C$JUSTIFY

C$JUSTIFY Built-In Subroutine Syntax

```
CALL "C$JUSTIFY" USING data-item, "justification-type"
~~~~             ~~~~~
```

Use `C$JUSTIFY` to left, right or center-justify an alphabetic, alphanumeric or numeric edited data-item. The optional *justification-type* argument indicates the type of the justification to be performed. Its value is interpreted as follows:

'`C`' the value will be centered

'`R`' the value will be right-justified, space-filled to the left

'`L`' the value will be left-justified, space-filled to the right

If it begins with anything else, or is absent, it will be treated as if it is present and begins with a capital '`R`'

8.2.8. C$MAKEDIR

C$MAKEDIR Built-In Subroutine Syntax

```
CALL "C$MAKEDIR" USING dir-path
~~~~             ~~~~~
```

With this routine you may create a new directory — the name of which is supplied as the *dir-path* argument (an alphanumeric literal or identifier).

Only the lowest-level directory (last) in the specified path can be created — all others must already exist. This subroutine will *not* behave as a `mkdir -p` (Unix) or `mkdir /p` (Windows).

The `RETURN-CODE` special register (see [Special Registers], page 265) will be set to the return code of the operation; the value will be either 0=Success or 128=failure.

8.2.9. C$NARG

C$NARG Built-In Subroutine Syntax

```
CALL "C$NARG" USING arg-count-result
~~~~          ~~~~~
```

This subroutine returns the number of arguments passed to the program that calls it back to in the numeric field *arg-count-result*. When called from within a user-defined function, a value of one (1) is returned if any arguments were passed to the function or a zero (0) otherwise.

When called from a main program, the returned value will always be 0.

8.2.10. C$PARAMSIZE

C$PARAMSIZE Built-In Subroutine Syntax

```
CALL "C$PARAMSIZE" USING argument-number
~~~~               ~~~~~
```

This subroutine returns the size (in bytes) of the subroutine argument supplied using the *argument-number* parameter (a numeric literal or data item).

The size is returned in the `RETURN-CODE` special register (see [Special Registers], page 265).

If the specified argument does not exist, or an invalid argument number is specified, a value of 0 is returned.

8.2.11. C$PRINTABLE

C$PRINTABLE Built-In Subroutine Syntax

```
CALL "C$PRINTABLE" USING data-item [ , char ]
~~~~               ~~~~~
```

The `C$PRINTABLE` subroutine converts the contents of the data-item specified as the first argument to printable characters. Those characters that are deemed printable (as defined by the character set used by *data-item*) will remain unchanged, while those that are NOT printable will be converted to the character specified as the second argument.

If no *char* argument is provided, a period ('`.`') will be used.

Note: CBL_GC_PRINTABLE replaces this although it is currently still supported for legacy reasons.

8.2.12. C$SLEEP

C$SLEEP Built-In Subroutine Syntax

```
CALL "C$SLEEP" USING seconds-to-sleep
~~~~           ~~~~~
```

`C$SLEEP` puts the program to sleep for the specified number of seconds and/or fractions of a second. The *seconds-to-sleep* argument may be a numeric literal or data item.

Sleep times less than 1 will be interpreted as 0, subject to the speed of the CPU and the O/S (Operating System) used, as well as the timing of the generated C code, which can immediately returns control to the calling program without any sleep delay.

When using a variable argument defined as 9(n)v9(m) where n is maximum seconds in 7 days, i.e., (60 x 60 x 24 x 7) = 604,800 (seconds) and m is at a point too fast for the CPU and O/S. In practice m should be 2 for a hundredth of a second but actual testing against the target CPU would be need.

The maximum time can be adjusted by the define MAX_SLEEP_TIME during compilation of the compiler [and no I do not know where it is in the codebase] e.g.:

```
/* maximum sleep time in seconds, currently 7 days */
#define MAX_SLEEP_TIME 3600*24*7
```

8.2.13. C$TOLOWER

C$TOLOWER Built-In Subroutine Syntax

```
CALL "C$TOLOWER" USING data-item, BY VALUE convert-length
~~~~             ~~~~~                ~~~~~
```

This routine will converts the *convert-length* (a numeric literal or data item) leading characters of *data-item* (an alphanumeric identifier) to lower-case.

The *convert-length* argument must be specified `BY VALUE` (see [CALL], page 293). Any characters in *data-item* after the *convert-length* point will remain unchanged.

If *convert-length* is negative or zero, no conversion will be performed.

8.2.14. C$TOUPPER

C$TOUPPER Built-In Subroutine Syntax

```
CALL "C$TOUPPER" USING data-item, BY VALUE convert-length
~~~~             ~~~~~               ~~~~~
```

This routine will converts the *convert-length* (a numeric literal or data item) leading characters of *data-item* (an alphanumeric identifier) to upper-case.

The *convert-length* argument must be specified `BY VALUE` (see [CALL], page 293). Any characters in *data-item* after the *convert-length* point will remain unchanged.

If *convert-length* is negative or zero, no conversion will be performed.

8.2.15. CBL_AND

CBL_AND Built-In Subroutine Syntax

```
CALL "CBL_AND" USING item-1, item-2, BY VALUE byte-length
 ~~~~           ~~~~~                 ~~~~~
```

Old Arg 1 Bit	Old Arg 2 Bit	New Arg 2 Bit
0	0	0
0	1	0
1	0	0
1	1	1

This subroutine performs a bit-by-bit logical `AND` operation between the left-most 8**byte-length* corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*. The truth table shown to the left documents the `AND` process.

The *item-1* argument may be an alphanumeric literal or a data item and *item-2* must be a data item. The length of both *item-1* and *item-2* must be at least 8**byte-length*.

The *byte-length* argument may be a numeric literal or data item, and must be specified using `BY VALUE` (see [CALL], page 293).

Any bits in *item-2* after the 8**byte-length* point will be unaffected.

A result of zero will be passed back in the `RETURN-CODE` special register (see [Special Registers], page 265).

8.2.16. CBL_CHANGE_DIR

CBL_CHANGE_DIR Built-In Subroutine Syntax

```
CALL "CBL_CHANGE_DIR" USING directory-path
~~~~                  ~~~~~
```

This routine makes *directory-path* (an alphanumeric literal or identifier) the current directory.

The return code of the operation, which will be either 0=Success or 128=failure, is returned in the `RETURN-CODE` special register (see [Special Registers], page 265).

The directory change remains in effect until the program terminates (in which the original current directory at the time the program was started will be automatically restored) or until another `CBL_CHANGE_DIR` or a `C$CHDIR` built-in system subroutine (see [C$CHDIR], page 545) is executed.

8.2.17. CBL_CHECK_FILE_EXIST

CBL_CHECK_FILE_EXIST Built-In Subroutine Syntax

```
CALL "CBL_CHECK_FILE_EXIST" USING file-path, file-info
~~~~                        ~~~~~
```

With this routine you may retrieve the size of the file specified as the *file-path* argument (an alphanumeric literal or identifier) and the date/time that file was last modified. File size information may not be available in the particular GnuCOBOL build / Operating System combination you are using and may therefore always be returned as zero.

The information is returned to the *file-info* argument, which is defined as the following 16-byte area:

```
01  file-info.
    05 File-Size-In-Bytes  PIC 9(18)  COMP.
    05 Mod-DD              PIC 9(2)   COMP.  *> Modification Date
    05 Mod-MO              PIC 9(2)   COMP.
    05 Mod-YYYY            PIC 9(4)   COMP.
    05 Mod-HH              PIC 9(2)   COMP.  *> Modification Time
    05 Mod-MM              PIC 9(2)   COMP.
    05 Mod-SS              PIC 9(2)   COMP.
    05 FILLER              PIC 9(2)   COMP.  *> Always 00
```

If the subroutine is successful, a value of 0 will be returned in the `RETURN-CODE` special register (see [Special Registers], page 265). Failure to retrieve the needed statistics on the file will cause a `RETURN-CODE` special register value of 35 to be passed back. Supplying less than two arguments will generate a 128 `RETURN-CODE` special register value.

8.2.18. CBL_CLOSE_FILE

CBL_CLOSE_FILE Built-In Subroutine Syntax

```
CALL "CBL_CLOSE_FILE" USING file-handle
 ~~~~                  ~~~~~
```

The CBL_CLOSE_FILE subroutine closes a byte stream file previously opened by either the CBL_OPEN_FILE built-in system subroutine (see [CBL_OPEN_FILE], page 590) or CBL_CREATE_FILE built-in system subroutine (see [CBL_CREATE_FILE], page 564) subroutines.

If the file defined by the *file-handle* argument (a PIC X(4) USAGE COMP-X data item) was opened for output, an implicit CBL_FLUSH_FILE built-in system subroutine (see [CBL_FLUSH_FILE], page 572) will be performed before the file is closed.

If the subroutine is successful, a value of 0 will be returned in the RETURN-CODE special register (see [Special Registers], page 265). Failure will cause a RETURN-CODE special register value of -1 to be passed back.

8.2.19. CBL_COPY_FILE

CBL_COPY_FILE Built-In Subroutine Syntax

```
CALL "CBL_COPY_FILE" USING src-file-path, dest-file-path
~~~~                 ~~~~~
```

Use this subroutine to copy file *src-file-path* to *dest-file-path* as if it were done via the `cp` (Unix/OSX) or `COPY` (Windows) command.

Both arguments may be alphanumeric literals or identifiers.

If the attempt to copy the file fails (for example, it or the destination directory doesn't exist), the `RETURN-CODE` special register (see [Special Registers], page 265) will be set to 128; on successful completion it will be set to 0.

8.2.20. CBL_CREATE_DIR

CBL_CREATE_DIR Built-In Subroutine Syntax

```
CALL "CBL_CREATE_DIR" USING dir-path
~~~~                  ~~~~~
```

With this routine you may create a new directory — the name of which is supplied as the *dir-path* argument (an alphanumeric literal or identifier).

Only the lowest-level directory (last) in the specified path can be created — all others must already exist. This subroutine will *not* behave as a `mkdir -p` (Unix) or `mkdir /p` (Windows).

The `RETURN-CODE` special register (see [Special Registers], page 265) will be set to the return code of the operation; the value will be either 0=Success or 128=failure.

8.2.21. CBL_CREATE_FILE

CBL_CREATE_FILE Built-In Subroutine Syntax

```
CALL "CBL_CREATE_FILE" USING file-path, 2, 0, 0, file-handle
~~~~                   ~~~~~
```

The `CBL_CREATE_FILE` subroutine creates the new file specified using the file-path argument and opens it for output as a byte-stream file usable by `CBL_WRITE_FILE` built-in system subroutine (see [CBL_WRITE_FILE], page 597).

Arguments 2, 3 and 4 should be coded as the constant values shown. `CBL_CREATE_FILE` is actually a special-case of the `CBL_OPEN_FILE` built-in system subroutine (see [CBL_OPEN_FILE], page 590) routine — see that routine for a description of the meanings of arguments 2, 3 and 4.

A *file-handle* (`PIC X(4) USAGE COMP-X`) will be returned, for use on any subsequent `CBL_WRITE_FILE` built-in system subroutine (see [CBL_WRITE_FILE], page 597) or `CBL_CLOSE_FILE` built-in system subroutine (see [CBL_CLOSE_FILE], page 561) calls.

The success or failure of the subroutine will be reported back in the `RETURN-CODE` special register (see [Special Registers], page 265), with a value of -1 indicating an invalid argument and a value of 0 indicating success.

8.2.22. CBL_DELETE_DIR

CBL_DELETE_DIR Built-In Subroutine Syntax

```
CALL "CBL_DELETE_DIR" USING dir-path
~~~~                  ~~~~~
```

This subroutine deletes an empty directory.

The only argument — *dir-path* (an alphanumeric literal or identifier) — is the name of the directory to be deleted.

Only the lowest-level directory (last) in the specified path will be deleted, and that directory must be empty to be deleted.

The `RETURN-CODE` special register (see [Special Registers], page 265) will be set to the return code of the operation; the value will be either 0=Success or 128=failure.

8.2.23. CBL_DELETE_FILE

CBL_DELETE_FILE Built-In Subroutine Syntax

```
CALL "CBL_DELETE_FILE" USING file-path
~~~~                   ~~~~~
```

This routine deletes the file specified by the *file-path* argument (an alphanumeric literal or identifier) just as if that were done using the `rm` (Unix/OSX) or `ERASE` (Windows) command.

If the attempt to delete the file fails (for example, it doesn't exist), the `RETURN-CODE` special register (see [Special Registers], page 265) will be set to 128; on successful completion it will be set to 0.

8.2.24. CBL_EQ

CBL_EQ Built-In Subroutine Syntax

```
CALL "CBL_EQ" USING item-1, item-2, BY VALUE byte-length
~~~~          ~~~~~                 ~~~~~
```

Old Arg 1 Bit	Old Arg 2 Bit	New Arg 2 Bit
0	0	1
0	1	0
1	0	0
1	1	1

This subroutine performs a bit-by-bit comparison between the left-most 8**byte-length* corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*. The truth table shown to the left documents the EQ process.

The *item-1* argument may be an alphanumeric literal or a data item and *item-2* must be a data item. The length of both *item-1* and *item-2* must be at least 8**byte-length*.

The *byte-length* argument may be a numeric literal or data item, and must be specified using `BY VALUE` (see [CALL], page 293).

Any bits in *item-2* after the 8**byte-length* point will be unaffected.

A result of zero will be passed back in the `RETURN-CODE` special register (see [Special Registers], page 265).

8.2.25. CBL_ERROR_PROC

CBL_ERROR_PROC Built-In Subroutine Syntax

```
CALL "CBL_ERROR_PROC" USING function, program-pointer
~~~~                  ~~~~~
```

This routine registers a general error-handling routine.

The *function* argument must be a numeric literal or a 32-bit binary data item (`USAGE BINARY-LONG`, for example) with a value of 0 or 1. A value of 0 means that you will be registering ("installing") an error procedure while a value of 1 indicates you're de-registering ("uninstalling") a previously-installed error procedure.

The *program-pointer* must be a data item with a `USAGE` (see [USAGE], page 232) of `PROGRAM-POINTER` containing the address of your error procedure. This item should be given a value using the `SET Program-Pointer` statement (see [SET Program-Pointer], page 383). If the error procedure is written in GnuCOBOL, it must be a subroutine, not a user-defined function.

A success (0) or failure (non-0) result will be passed back in the `RETURN-CODE` special register (see [Special Registers], page 265).

A custom error procedure will trigger when a runtime error condition is encountered. An error procedure may be registered by a main program or a subprogram, but regardless of from where it was registered, it applies to the overall program compilation group and will trigger when a runtime error occurs anywhere in the executable program. If the error procedure was defined by a subprogram, that program must be loaded at the time the error procedure is executed.

An error procedure may be used to take whatever actions might be warranted to display additional information or to gracefully close down work in progress, but it cannot *prevent* the termination of program execution; should the error procedure not issue its own `STOP RUN`, control will return back to the standard error routine when the error procedure exits.

The code within the handler will be executed and — once the handler issues a `return`, if it was written in C, or an `EXIT PROGRAM` statement (see [EXIT], page 328) or `GOBACK` statement, if it was written in GnuCOBOL, the system-standard error handling routine will be executed.

Only one user-defined error procedure may be in effect at any time.

The following is a sample GnuCOBOL program that registers an error procedure. The output of that program is shown as well. As as you can see, the error handler's messages appear followed by the standard GnuCOBOL message.

```
1.      IDENTIFICATION DIVISION.
2.      PROGRAM-ID. DemoERRPROC.
3.      ENVIRONMENT DIVISION.
4.      DATA DIVISION.
5.      WORKING-STORAGE SECTION.
```

```
6.      01  Err-Proc-Address            USAGE PROGRAM-POINTER.
7.      PROCEDURE DIVISION.
8.      S1.
9.          DISPLAY 'Program is starting'
10.         SET Err-Proc-Address TO ENTRY 'ErrProc'
11.         CALL 'CBL_ERROR_PROC' USING 0, Err-Proc-Address
12.         CALL 'Tilt' *> THIS DOESN'T EXIST!!!!
13.         DISPLAY 'Program is stopping'
14.         STOP RUN
15.         .
16.     END PROGRAM DemoERRPROC.
17.
18.     IDENTIFICATION DIVISION.
19.     PROGRAM-ID. ErrProc.
20.     PROCEDURE DIVISION.
21.     000-Main.
22.         DISPLAY 'Error: ' FUNCTION EXCEPTION-LOCATION
23.         DISPLAY '       ' FUNCTION EXCEPTION-STATEMENT
24.         DISPLAY '       ' FUNCTION EXCEPTION-FILE
25.         DISPLAY '       ' FUNCTION EXCEPTION-STATUS
26.         DISPLAY '*** Returning to Standard Error Routine ***'
27.         EXIT PROGRAM
28.         .
29.     END PROGRAM ErrProc.
```

When executed, this sample program generates the following console output.

```
E:\Programs\Demos>demoerrproc
Program is starting
Error: DemoERRPROC; S1; 12
       CALL
       00
       EC-PROGRAM-NOT-FOUND
*** Returning to Standard Error Routine ***
DEMOERRPROC.cbl: 27: libcob: Cannot find module 'Tilt'

E:\Programs\Demos>
```

8.2.26. CBL_EXIT_PROC

CBL_EXIT_PROC Built-In Subroutine Syntax

```
CALL "CBL_EXIT_PROC" USING function, program-pointer
~~~~                 ~~~~~
```

This routine registers a general exit-handling routine.

The *function* argument must be a numeric literal or a 32-bit binary data item (`USAGE BINARY-LONG`, for example) with a value of 0 or 1. A value of 0 means that you will be registering ("installing") an exit procedure while a value of 1 indicates you're deregistering ("uninstalling") a previously-installed exit procedure.

The *program-pointer* must be a data item with a `USAGE` (see [USAGE], page 232) of `PROGRAM-POINTER` containing the address of your exit procedure.

A success (0) or failure (non-0) result will be passed back in the `RETURN-CODE` special register (see [Special Registers], page 265).

An exit procedure, once registered, will trigger whenever a `STOP RUN` statement (see [STOP], page 399) or a `GOBACK` statement (see [GOBACK], page 334) is executed anywhere in the program. The exit procedure may execute whatever code is desired to undertake an orderly shut down of the program. Once the exit procedure terminates by executing an `EXIT PROGRAM` statement (see [EXIT], page 328) or a `GOBACK` statement, the system-standard program termination routine will be executed.

Only one user-defined exit procedure may be in effect at any time.

The following is a sample GnuCOBOL program that registers an exit procedure. The output of that program is shown as well.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. demoexitproc.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Exit-Proc-Address           USAGE PROGRAM-POINTER.
PROCEDURE DIVISION.
000-Register-Exit-Proc.
    SET Exit-Proc-Address TO ENTRY "ExitProc"
    CALL "CBL_EXIT_PROC" USING 0, Exit-Proc-Address
    IF RETURN-CODE NOT = 0
        DISPLAY 'Error: Could not register Exit Procedure'
    END-IF
    .
099-Now-Test-Exit-Proc.
    DISPLAY
        'Executing a STOP RUN...'
    END-DISPLAY
    GOBACK.
```

```
END PROGRAM demoexitproc.

IDENTIFICATION DIVISION.
PROGRAM-ID. ExitProc.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Display-Date               PIC XXXX/XX/XX.
01  Display-Time               PIC XX/XX/XX.
01  Now                        PIC X(8).
01  Today                      PIC X(8).
PROCEDURE DIVISION.
000-Main.
    DISPLAY '*** STOP RUN has been executed ***'
    ACCEPT Today FROM DATE YYYYMMDD
    ACCEPT Now   FROM TIME
    MOVE Today TO Display-Date
    MOVE Now   TO Display-Time
    INSPECT Display-Time REPLACING ALL '/' BY ':'
    DISPLAY '***    ' Display-Date '  ' Display-Time '    ***'
    GOBACK.
END PROGRAM ExitProc.
```

8.2.27. CBL_FLUSH_FILE

CBL_FLUSH_FILE Built-In Subroutine Syntax

```
CALL "CBL_FLUSH_FILE" USING file-handle
~~~~                  ~~~~~
```

In Micro Focus COBOL, calling this subroutine flushes any as-yet unwritten buffers for the (output) file whose file-handle is specified as the argument to disk.

This routine is non-functional in GnuCOBOL. It exists only to provide compatibility for applications that may have been developed for Micro Focus COBOL.

8.2.28. CBL_GC_FORK

CBL_GC_FORK Built-In Subroute Syntax

```
CALL "CBL_GC_FORK" USING Child-PID
~~~~               ~~~~~
```

CBL_GC_FORK allows you to fork the current COBOL process to a new one.

The current content of the process's storage (including LOCAL-STORAGE) will be identical, any file handles get invalid in the new process, positions and file and record locks are only available to the original process.

This system routine is not available on Windows (exception: GCC on Cygwin).

Parameters: none

Returns: `pid` (the child process gets '0' returned, the calling process gets the `pid` of the created child).

Negative values are returned for system dependant error codes and -1 if the function is not available on the current system.

`CBL_GC_FORK` allows you to fork the current COBOL process to a new one. The current content of the process' storage (including `LOCAL-STORAGE`) will be identical, any file handles get invalid in the new process, positions and file / record locks are only available to the original process. This system routine is not available on Windows (exception: `gcc` on Cygwin). Parameters: none Returns: `pid` (the child process gets 0 returned, the calling process gets the `pid` of the created children). Negative values are returned for system dependant error codes and -1 if the function is not available on the current system.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. prog.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  CHILD-PID      PIC S9(9) BINARY.
01  WAIT-STS       PIC S9(9) BINARY.
PROCEDURE DIVISION.
    CALL     "CBL_GC_FORK" RETURNING CHILD-PID END-CALL
    EVALUATE TRUE
             WHEN CHILD-PID = ZERO
                  PERFORM CHILD-CODE
             WHEN CHILD-PID > ZERO
                  PERFORM PARENT-CODE
             WHEN CHILD-PID = -1
                  DISPLAY 'CBL_GC_FORK is not available on the current'
                  ' system!'
                  PERFORM CHILD-CODE
                  MOVE 0 TO CHILD-PID
                  PERFORM PARENT-CODE
```

```
            WHEN OTHER
                 MULTIPLY -1 BY CHILD-PID END-MULTIPLY
                 DISPLAY 'CBL_GC_FORK returned system error: ' CHILD-PID
    END-EVALUATE
    STOP     RUN.
CHILD-CODE.
    CALL     "C$SLEEP" USING 1 END-CALL
    DISPLAY  "Hello, I am the child"
    MOVE     2 TO RETURN-CODE.
PARENT-CODE.
    DISPLAY  "Hello, I am the parent"
    CALL     "CBL_GC_WAITPID" USING CHILD-PID RETURNING WAIT-STS
    MOVE     0 TO RETURN-CODE
    EVALUATE TRUE
            WHEN WAIT-STS >= 0
                 DISPLAY 'Child ended with status: ' WAIT-STS
            WHEN WAIT-STS = -1
                 DISPLAY 'CBL_GC_WAITPID is not available on the '
                 'current system!'
            WHEN WAIT-STS < -1
                 MULTIPLY -1 BY WAIT-STS END-MULTIPLY
                 DISPLAY 'CBL_GC_WAITPID returned system error: ' WAIT-STS
    END-EVALUATE.
```

8.2.29. CBL_GC_GETOPT

CBL_GC_GETOPT Built-In Subroutine Syntax

```
CALL "CBL_GC_GETOPT" USING BY REFERENCE SHORTOPTIONS LONGOPTIONS LONGIND
~~~~                 ~~~~~
                           BY VALUE LONG-ONLY
                           BY REFERENCE RETURN-CHAR OPT-VAL
```

CBL_GC_GETOPT adapts the well-known option parser, getopt, to GnuCOBOL.

The usage of this system routine is described by the following example.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG.
DATA DIVISION.
WORKING-STORAGE SECTION.
78  SHORTOPTIONS VALUE "jkl".
01  LONGOPTIONS.
    05 OPTIONRECORD OCCURS 2 TIMES.
       10  OPTIONNAME   PIC X(25).
       10  HAS-VALUE    PIC 9.
       10  VALPOINT     POINTER VALUE NULL.
       10  RETURN-VALUE PIC X(4).
01  LONGIND             PIC 99.
01  LONG-ONLY           PIC 9 VALUE 1.
01  RETURN-CHAR         PIC X(4).
01  OPT-VAL             PIC X(10).
01  COUNTER             PIC 9 VALUE 0.
```

We first need to define the necessary fields for getopt's shortoptions, longoptions, longoption index (longind), long-only-option (long-only) and also the fields for return values return-char and opt-val (arbitrary size with trimming, see return codes).

The shortoptions are written down as an alphanumeric field (i.e., a string with arbitrary size) as follows:

```
"ab:c::d"
```

This means we want getopt to look for short options named 'a', 'b', 'c' or 'd', require an option value for 'b', and accept an optional one for 'c'.

The longoptions are defined as a table of records with oname, has-value, valpoint and val.[1]

oname defines the name of a longoption. has-value defines if an option value is demanded (has-val = 1), optional (has-val = 2) or not required (has-val = 0).

valpoint is a pointer used to specify an address to save getopt's return value to. The pointer is optional. If it is NULL, getopt returns a value as usual. If you use the pointer it

[1] Say what? the discussion and code seem to have diverged.

has to point to a PIC X(4) field. The field val is a PIC X(4) character which is returned if the longoption was recognized.

The longoption structure is immutable! You can only vary the number of records.

Now we have the tools to run CBL_GC_GETOPT within the procedure division.

```
PROCEDURE DIVISION.
     MOVE     "version" to OPTIONNAME (1).
     MOVE     0 TO HAS-VALUE (1).
     MOVE     'V' TO RETURN-VALUE (1).
     MOVE     "verbose" TO OPTIONNAME (2).
     MOVE     0 TO HAS-VALUE (2).
     MOVE     'V' TO RETURN-VALUE (2).
     PERFORM  WITH TEST AFTER UNTIL RETURN-CODE = -1
              CALL 'CBL_GC_GETOPT' USING
                  BY REFERENCE SHORTOPTIONS LONGOPTIONS LONGIND
                  BY VALUE LONG-ONLY
                  BY REFERENCE RETURN-CHAR OPT-VAL
              END-CALL
              DISPLAY RETURN-CHAR END-DISPLAY
              DISPLAY OPT-VAL END-DISPLAY
     END-PERFORM
     STOP RUN.
```

The example shows how we initialize all parameters and call the routine until CBL_GC_GETOPT runs out of options and returns -1.

return-char might contain the following regular character if an option was recognized:

?	undefined or ambiguous option
1	non-option (only if first byte of so is '-')
0	valpoint != NULL and we are writing the return value to the specified address
-1	no more options (or reach the first non-option if first byte of shortoptions is '+')

The return-codes of CBL_GC_GETOPT are:

1	a non-option (only if first byte of so is '-')
0	valpoint != NULL and we are writing the return value to the specified address
-1	no more options (or reach the first non-option if first byte of shortoptions is '+')
2	truncated option value in opt-val (because opt-val was too small)
3	a regular answer from getopt

8.2.30. CBL_GC_HOSTED

CBL_GC_HOSTED Built-In Subroutine Syntax

```
CALL "CBL_GC_HOSTED" USING ARG-1  ARG-2
~~~~                 ~~~~~

Note replaces CBL_OC_HOSTED which is kept as a legacy item.
```

CBL_GC_HOSTED provides access to the following C hosted variables:

argc binary-long by value

argv pointer to char **

stdin, stdout, stderr
pointer

errno giving address of errno in pointer to binary-long, use based for more

Direct access and conditional access to the following variables:

tzname pointer to pointer to array of two char pointers

timezone C long, will be seconds west of UTC

daylight C int, will be 1 during daylight savings

The system will need HAVE TIMEZONE defined for these to return anything meaningful. Attempts made when they are not available will return 1 from CBL GC HOSTED.

It returns 0 when match, 1 on failure, case matters as does length, "arg" won't match.

The usage of this system routine is described by the following example.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. HOSTED.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  Argc        BINARY-LONG.
 01  Argv        POINTER.
 01  Stdin       POINTER.
 01  Stdout      POINTER.
 01  Stderr      POINTER.
 01  Errno       POINTER.
 01  Err         BINARY-LONG BASED.
 01  Domain      FLOAT-LONG VALUE 3.0.
 01  Tzname      POINTER.
 01  Tznames     POINTER BASED.
     05  Tzs     POINTER OCCURS 2.
 01  Timezone    BINARY-LONG.
```

```
 01  Daylight    BINARY-SHORT.
*>
 PROCEDURE DIVISION.
     call     "CBL_GC_HOSTED" using stdin "stdin"
     display  "stdin : " stdin
     call     "feof" using by value stdin
     display  "feof stdin : " return-code
     call     "CBL_GC_HOSTED" using stdout "stdout"
     display  "stdout : " stdout
     call     "fprintf" using by value stdout by content "Hello" & x"0a"
     call     "CBL_GC_HOSTED" using stderr "stderr"
     display  "stderr : " stderr
     call     "fprintf" using by value stderr by content "on err" & x"0a"
     call     "CBL_GC_HOSTED" using argc "argc"
     display  "argc : " argc
     call     "CBL_GC_HOSTED" using argv "argv"
     display  "argv : " argv
     call     "args" using by value argc argv
     call     "CBL_GC_HOSTED" using errno "errno"
     display  "&errno : " errno
     set      address of err to errno
     display  "errno : " err
     call     "acos" using by value domain
     display  "errno after acos(3.0): " err ", EDOM is 33"
     call     "CBL_GC_HOSTED" using argc "arg"
     display  "'arg' lookup : " return-code
     call     "CBL_GC_HOSTED" using null "argc"
     display  "null with argc : " return-code
     display  "argc is still : " argc
*> the following only returns zero if the system has HAVE_TIMEZONE set
     call     "CBL_GC_HOSTED" using daylight "daylight "
     display  "'timezone' lookup : " return-code
     if       return-code not = 0
              display "system doesn't has timezone"
     else
              display "timezone is : " timezone
              call "CBL_GC_HOSTED" using daylight "daylight "
              display "'daylight' lookup : " return-code
              display "daylight is : " daylight
              set environment "TZ" to "PST8PDT"
              call static "tzset" returning omitted on exception
                        continue end-call
              call "CBL_GC_HOSTED" using tzname "tzname"
              display "'tzname' lookup : " return-code
*> tzs(1) will point to z"PST" and tzs(2) to z"PDT"
              if   return-code equal 0 and tzname not equal null then
```

```
                    set address of tznames to tzname
                    if   tzs(1) not equal null then
                         display "tzs #1 : " tzs(1)
                    end-if
                    if   tzs(2) not equal null then
                         display "tzs #2 : " tzs(2)
                    end-if
               end-if
          end-if
          goback.
      end program hosted.
```

Note that the legacy name of this routine that starts with `CBL_OC` is deprecated, as is `NANOSLEEP` but will still work. It is recommended that all library routines names starting with `CBL_OC` are replaced with `CBL_GC` to minimise issues.

8.2.31. CBL_GC_NANOSLEEP

CBL_GC_NANOSLEEP Built-In Subroutine Syntax

```
CALL "CBL_GC_NANOSLEEP" USING nanoseconds-to-sleep
~~~~                    ~~~~~

Note replaces CBL_OC_NANOSLEEP which is kept as a legacy item.
```

This subroutine puts the program to sleep for the specified number of nanoseconds.

The effective granularity of *nanoseconds-to-sleep* values will depend upon the granularity of the system clock your computer is using and the timing granularity of the operating system that computer is running.

For example, you will not expect to see any difference between values of 1, 100, 500 or 1000, but you should see a difference between values such as 250000000 and 500000000.

The *nanoseconds-to-sleep* argument is a numeric literal or data item.

There are one *billion* nanoseconds in a second, so if you wanted to put the program to sleep for 1/4 second you'd use a *nanoseconds-to-sleep* value of 250000000.

Note that the legacy name of this routine starts with "CBL_OC" is deprecated, as is `HOSTED`, but will still work. It is recommended that all library routines names starting with "CBL_OC" are replaced with "CBL_GC" to minimise issues.

8.2.32. CBL_GC_PRINTABLE

CBL_GC_PRINTABLE Built-In Subroutine Syntax

```
CALL "CBL_GC_PRINTABLE" USING data-item [ , char ]
~~~~                    ~~~~~

Note replaces C$PRINTABLE which is kept as a legacy item.
```

The `CBL_GC_PRINTABLE` subroutine converts the contents of the data-item specified as the first argument to printable characters.

Those characters that are deemed printable (as defined by the character set used by *data-item*) will remain unchanged, while those that are *not* printable will be converted to the character specified as the second argument.

If no *char* argument is provided, a period ('`.`') will be used.

8.2.33. CBL_GC_WAITPID

CBL_GC_WAITPID Built-In Subroutine Syntax

```
CALL "CBL_GC_WAITPID" USING ARG-1
~~~~                  ~~~~~
                      RETURNING RET-STATUS
                      ~~~~~~~~~
```

CBL_GC_WAITPID allows you to wait until another system process ended.

Additionally you can check the process's return code.

Parameters: none

Returns: function-status / child-status

Negative values are returned for system dependant error codes and -1 if the function is not available on the current system.

```
CALL     "CBL_GC_WAITPID" USING CHILD-PID RETURNING WAIT-STS
MOVE     0 TO RETURN-CODE
DISPLAY  'CBL_GC_WAITPID ended with status: ' WAIT-STS
```

8.2.34. CBL_GET_CSR_POS

CBL_GET_CSR_POS Built-In Subroutine Syntax

```
CALL "CBL_GET_CSR_POS" USING cursor-locn-buffer
~~~~                   ~~~~~
```

This subroutine will retrieve the current cursor location on the screen, returning a 2-byte value into the supplied *cursor-locn-buffer*. The first byte of *cursor-locn-buffer* will receive the current line (row) location while the second receives the current column location.

The returned location data will be in binary form, and will be based upon starting values of 0, meaning that if the cursor is located at line 15, column 12 at the time this routine is called, a value of (14,11) will be returned.

The following is a typical *cursor-locn-buffer* definition:

```
01  CURSOR-LOCN-BUFFER.
    05 CURSOR-LINE          USAGE BINARY-CHAR.
    05 CURSOR-COLUMN        USAGE BINARY-CHAR.
```

Values of 1 (Line) and 1 (column) will be returned if GnuCOBOL was not generated to include screen I/O.

8.2.35. CBL_GET_CURRENT_DIR

CBL_GET_CURRENT_DIR Built-In Subroutine Syntax

```
CALL "CBL_GET_CURRENT_DIR" USING BY VALUE 0,
                                 BY VALUE length,
                                 BY REFERENCE buffer
```

This retrieves the fully-qualified pathname of the current directory, saving up to *length* characters of that name into *buffer*.

The first argument is unused, but must be specified. It must be specified `BY VALUE` (see [CALL], page 293).

The *length* argument must be specified `BY VALUE`. The *buffer* argument must be specified `BY REFERENCE`.

The value specified for the *length* argument (a numeric literal or data item) should not exceed the actual length of *buffer* argument.

If the value specified for the *length* argument is LESS THAN the actual length of *buffer* argument, the current directory path will be left-justified and space filled within the first *length* bytes of *buffer* — any bytes in *buffer* after that point will be unchanged.

If the routine is successful, a value of 0 will be returned to the `RETURN-CODE` special register (see [Special Registers], page 265). If the routine failed because of a problem with an argument (such as a negative or 0 length), a value of 128 will result. Finally, if the 1st argument value is anything but zero, the routine will fail with a 129 value.

8.2.36. CBL_GET_SCR_SIZE

CBL_GET_SCR_SIZE Built-In Subroutine Syntax

```
CALL "CBL_GET_SCR_SIZE" USING no-of-lines, no-of-cols
~~~~                    ~~~~~
```

Use this subroutine to retrieve the current console screen size.

When the system is running in a windowed environment, this will be the sizing of the console window in which the program is executing. When the system is not running a windowing environment, the physical console screen attributes will be returned. In environments such as a Windows console window, where the logical size of the window may far exceed that of the physical console window, the size returned will be that of the physical console window. Two one-byte values will be returned — the first will be the current number of lines (rows) while the second will be the number of columns.

The returned size data will be in binary form.

The following are typical *no-of-lines* and *no-of-columns* definitions:

```
01  NO-OF-LINES             USAGE BINARY-CHAR.
01  NO-OF-COLUMNS           USAGE BINARY-CHAR.
```

GnuCOBOL run-time screen management must have been initialized prior to CALLing this routine in order to receive meaningful values. This means that a `DISPLAY data-item` statement (see [DISPLAY data-item], page 311) or a `ACCEPT data-item` statement (see [ACCEPT data-item], page 272) must have been executed prior to executing the `CALL` statement.

Zero values will be returned if the screen has not been initialized and values of 24 (lines) and 80 (columns) will be returned if GnuCOBOL was not generated to include screen I/O.

8.2.37. CBL_IMP

CBL_IMP Built-In Subroutine Syntax

```
CALL "CBL_IMP" USING item-1, item-2, BY VALUE byte-length
~~~~           ~~~~~                 ~~~~~
```

Old Arg 1 Bit	Old Arg 2 Bit	New Arg 2 Bit
0	0	1
0	1	1
1	0	0
1	1	1

This subroutine performs a bit-by-bit logical *implies* process between the left-most 8**byte-length* corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*. The truth table shown to the left documents the `IMP` process.

The *item-1* argument may be an alphanumeric literal or a data item and *item-2* must be a data item. The length of both *item-1* and *item-2* must be at least 8**byte-length*.

The *byte-length* argument may be a numeric literal or data item, and must be specified using `BY VALUE` (see [CALL], page 293).

Any bits in *item-2* after the 8**byte-length* point will be unaffected.

A result of zero will be passed back in the `RETURN-CODE` special register (see [Special Registers], page 265).

8.2.38. CBL_NIMP

CBL_NIMP Built-In Subroutine Syntax

```
CALL "CBL_NIMP" USING item-1, item-2, BY VALUE byte-length
 ~~~~            ~~~~~                 ~~~~~
```

Old Arg 1 Bit	Old Arg 2 Bit	New Arg 2 Bit
0	0	0
0	1	0
1	0	1
1	1	0

This subroutine performs the *negation* of a bit-by-bit logical *implies* process between the left-most 8**byte-length* corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*. The truth table shown to the left documents the `NIMP` process.

The *item-1* argument may be an alphanumeric literal or a data item and *item-2* must be a data item. The length of both *item-1* and *item-2* must be at least 8**byte-length*.

The *byte-length* argument may be a numeric literal or data item, and must be specified using `BY VALUE` (see [CALL], page 293).

Any bits in *item-2* after the 8**byte-length* point will be unaffected.

A result of zero will be passed back in the `RETURN-CODE` special register (see [Special Registers], page 265).

8.2.39. CBL_NOR

CBL_NOR Built-In Subroutine Syntax

```
CALL "CBL_NOR" USING item-1, item-2, BY VALUE byte-length
 ~~~~           ~~~~~                 ~~~~~
```

Old Arg 1 Bit	Old Arg 2 Bit	New Arg 2 Bit
0	0	1
0	1	0
1	0	0
1	1	0

This subroutine performs the negation of a bit-by-bit logical *or'* process between the left-most 8**byte-length* corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*. The truth table shown to the left documents the `NOR` process.

The *item-1* argument may be an alphanumeric literal or a data item and *item-2* must be a data item. The length of both *item-1* and *item-2* must be at least 8**byte-length*.

The *byte-length* argument may be a numeric literal or data item, and must be specified using `BY VALUE` (see [CALL], page 293).

Any bits in *item-2* after the 8**byte-length* point will be unaffected.

A result of zero will be passed back in the `RETURN-CODE` special register (see [Special Registers], page 265).

8.2.40. CBL_NOT

CBL_NOT Built-In Subroutine Syntax

```
CALL "CBL_NOT" USING item-1, BY VALUE byte-length
~~~~           ~~~~~         ~~~~~
```

This subroutine "flips" the left-most 8**byte-length* bits of *item-1*, changing 0 bits to 1, and 1 bits to 0. The changes are made directly im *item-1*.

The *item-1* argument must be a data item. The length of *item-1* must be at least 8**byte-length*.

The *byte-length* argument may be a numeric literal or data item, and must be passed using `BY VALUE` (see [CALL], page 293).

Any bits in *item-1* after the 8**byte-length* point will be unaffected.

A result of zero will be passed back in the `RETURN-CODE` special register (see [Special Registers], page 265).

8.2.42. CBL_OPEN_FILE

CBL_OPEN_FILE Built-In Subroutine Syntax

```
CALL "CBL_OPEN_FILE" USING file-path, access-mode, 0, 0, handle
~~~~                 ~~~~~
```

This routine opens an existing file for use as a byte-stream file usable by CBL_WRITE_FILE or CBL_READ_FILE.

The *file-path* argument is an alphanumeric literal or data-item.

The *access-mode* argument is a numeric literal or data item with a `PIC X USAGE COMP-X` (or `USAGE BINARY-CHAR`) definition; it specifies how you wish to use the file, as follows:

`1` input (read-only)

`2` output (write-only)

`3` input and/or output

The third and fourth arguments would specify a locking mode and device specification, respectively, but they're not implemented in GnuCOBOL (currently, at least) — just specify each as 0.

The final argument (*handle*) is a `PIC X(4) USAGE COMP-X` item that will receive the handle to the file. That handle is used on all other byte-stream functions to reference this specific file.

A `RETURN-CODE` special register (see [Special Registers], page 265) value of -1 indicates an invalid argument, while a value of 0 indicates success. A value of 35 means the file does not exist.

8.2.43. CBL_OR

CBL_OR Built-In Subroutine Syntax

```
CALL "CBL_OR" USING item-1, item-2, BY VALUE byte-length
 ~~~~           ~~~~~                       ~~~~~
```

Old Arg 1 Bit	Old Arg 2 Bit	New Arg 2 Bit
0	0	0
0	1	1
1	0	1
1	1	1

This subroutine performs a bit-by-bit logical *or* process between the left-most 8**byte-length* corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*. The truth table shown to the left documents the `OR` process.

The *item-1* argument may be an alphanumeric literal or a data item and *item-2* must be a data item. The length of both *item-1* and *item-2* must be at least 8**byte-length*.

The *byte-length* argument may be a numeric literal or data item, and must be specified using `BY VALUE` (see [CALL], page 293).

Any bits in *item-2* after the 8**byte-length* point will be unaffected.

A result of zero will be passed back in the `RETURN-CODE` special register (see [Special Registers], page 265).

8.2.44. CBL_READ_FILE

CBL_READ_FILE Built-In Subroutine Syntax

```
CALL "CBL_READ_FILE" USING handle, offset, nbytes, flag, buffer
~~~~                 ~~~~~
```

This routine reads *nbytes* of data starting at byte number *offset* from the byte-stream file defined by *handle* into *buffer*.

The *handle* argument (`PIC X(4) USAGE COMP-X`) must have been populated by a prior call to `CBL_OPEN_FILE` built-in system subroutine (see [CBL_OPEN_FILE], page 590).

The *offset* argument (`PIC X(8) USAGE COMP-X`) defines the location in the file of the first byte to be read. The first byte of a file is byte offset 0 and MUST be preset to zero for first use.

The *nbytes* argument (`PIC X(4) USAGE COMP-X`) specifies how many bytes (maximum) will be read. If the *flag* argument is specified as 128, the size of the file (in bytes) will be returned into the file offset argument (argument 2) upon completion. Not all operating system/GnuCOBOL environments may be able to retrieve file sizes in such cases, a value of zero will be returned. The only other valid value for flags is 0. This argument may be specified either as a numeric literal or as a `PIC X USAGE COMP-X` data item.

Upon completion, the `RETURN-CODE` special register (see [Special Registers], page 265) will be set to 0 if the read was successful or to 10 if an "end-of-file" condition occurred. If a value of -1 is returned, a problem was identified with the subroutine arguments.

8.2.45. CBL_READ_KBD_CHAR

CBL_READ_KBD_CHAR Build-In Subroutine Syntax

```
CALL "CBL_READ_KBD_CHAR" USING char RETURNING status-code.
~~~~                     ~~~~~      ~~~~~~~~~
```

Waits until a character is typed from the terminal and then read it with no echo.

Parameters: char `PIC X`. Receives the character that was typed, in ASCII.

status-code `PIC XX COMP-5`.

If `RETURNING` is not used the `RETURN-CODE` special register receives the status-code where zero is success and non-zero it is not.

[Above information taken from MF WB manual].

8.2.46. CBL_RENAME_FILE

CBL_RENAME_FILE Built-In Subroutine Syntax

```
CALL "CBL_RENAME_FILE" USING old-file-path, new-file-path
~~~~                   ~~~~~
```

You may use this subroutine to rename a file.

The file specified by *old-file-path* will be "renamed" to the name specified as *new-file-path*. Each argument may be an alphanumeric literal or data item.

Despite what the name of this routine might make you believe, this routine is more than just a simple "rename" — it will actually move the file supplied as the 1st argument to the file specified as the 2nd argument. Think of it as a two-step sequence, first copying the *old-file-path* file to the *new-file-path* file and then a second step where the *old-file-path* is deleted.

If the attempt to move the file fails (for example, it doesn't exist), the `RETURN-CODE` special register (see [Special Registers], page 265) will be set to 128; on successful completion it will be set to 0.

8.2.47. CBL_SET_CSR_POS

CBL_SET_CSR_POS Build-In Subroutine Syntax

```
CALL "CBL_SET_CSR_POS" USING cursor-locn-buffer
~~~~                   ~~~~~
```

Set current cursor position on terminal.

This subroutine will set the cursor location on the screen, using a 2-byte value into the supplied *cursor-locn-buffer*. The first byte of *cursor-locn-buffer* is for the line (row) location while the second sets the column location.

The two byte data block must be in binary form, and will be based upon starting values of 0, meaning that if the routine is called with a value of (14,11) cursor will be located at line 15, column 12.

The following is a typical *cursor-locn-buffer* definition:

```
01  CURSOR-LOCN-BUFFER.
    05 CURSOR-LINE          USAGE BINARY-CHAR.
    05 CURSOR-COLUMN        USAGE BINARY-CHAR.
```

8.2.48. CBL_TOLOWER

CBL_TOLOWER Built-In Subroutine Syntax

```
CALL "CBL_TOLOWER" USING data-item, BY VALUE convert-length
~~~~               ~~~~~               ~~~~~
```

This routine will convert the first *convert-length* (a numeric literal or data item) characters of *data-item* (an alpha-numeric identifier) to lower-case.

The *convert-length* argument must be specified `BY VALUE` (see [CALL], page 293). It specifies how many (leading) characters in data-item will be converted — any characters after that will remain unchanged.

If *convert-length* is negative or zero, no conversion will be performed.

8.2.49. CBL_TOUPPER

CBL_TOUPPER Built-In Subroutine Syntax

```
CALL "CBL_TOUPPER" USING data-item, BY VALUE convert-length
~~~~               ~~~~~               ~~~~~
```

This routine will convert the first *convert-length* (a numeric literal or data item) characters of *data-item* (an alpha-numeric identifier) to upper-case.

The *convert-length* argument must be specified `BY VALUE` (see [CALL], page 293). It specifies how many (leading) characters in data-item will be converted — any characters after that will remain unchanged.

If *convert-length* is negative or zero, no conversion will be performed.

8.2.50. CBL_WRITE_FILE

CBL_WRITE_FILE Built-In Subroutine Syntax

```
CALL "CBL_WRITE_FILE" USING handle, offset, nbytes, 0, buffer
~~~~                  ~~~~~
```

This routine writes *nbytes* of data from *buffer* to the byte-stream file defined by *handle* starting at byte number *offset* within the file.

The *handle* argument (`PIC X(4) USAGE COMP-X`) must have been populated by a prior call to CBL_OPEN_FILE. The offset argument (`PIC X(4) USAGE COMP-X`) defines the location in the file of the first byte to be written to. The first byte of a file is byte offset 0.

The *nbytes* argument (`PIC X(4) USAGE COMP-X`) specifies how many bytes (maximum) will be written.

Currently, the only allowable value for the flags argument is 0. This argument may be specified either as a numeric literal or as a `PIC X(1) USAGE COMP-X` data item.

Upon completion, the `RETURN-CODE` special register (see [Special Registers], page 265) will be set to 0 if the write was successful or to 30 if an I/O error condition occurred. If a value of -1 is returned, a problem was identified with the subroutine arguments.

8.2.51. CBL_XOR

CBL_XOR Built-In Subroutine Syntax

```
CALL "CBL_XOR" USING item-1, item-2, BY VALUE byte-length
 ~~~~           ~~~~~                 ~~~~~
```

Old Arg 1 Bit	Old Arg 2 Bit	New Arg 2 Bit
0	0	0
0	1	1
1	0	1
1	1	0

This subroutine performs a bit-by-bit logical *exclusive or* process between the left-most 8**byte-length* corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*. The truth table shown to the left documents the `XOR` process.

The *item-1* argument may be an alphanumeric literal or a data item and *item-2* must be a data item. The length of both *item-1* and *item-2* must be at least 8**byte-length*.

The *byte-length* argument may be a numeric literal or data item, and must be specified using `BY VALUE` (see [CALL], page 293).

Any bits in *item-2* after the 8**byte-length* point will be unaffected.

A result of zero will be passed back in the `RETURN-CODE` special register (see [Special Registers], page 265).

8.2.52. SYSTEM

SYSTEM Built-In Subroutine Syntax

```
CALL "SYSTEM" USING command
~~~~          ~~~~~
```

This subroutine submits *command* (an alphanumeric literal or data item) to a command shell for execution as if it were typed into a console/terminal window.

A shell will be opened subordinate to the GnuCOBOL program issuing the call to `SYSTEM`.

Output from the command (if any) will appear in the command window in which the GnuCOBOL program was executed.

On a Unix system, the shell environment will be established using the default shell program. This is also true when using a GnuCOBOL build created with and for OSX or the Cygwin Unix emulator.

With native Windows Windows/MinGW builds, the shell environment will be the Windows console window command processor (usually `cmd.exe`) appropriate for the version of Windows you're using.

To trap output from the executed command and process it within the GnuCOBOL program, use a redirection ('`>`') to send the command output to a temporary file which you read from within the program once control returns.

The exit status of the executed command will be available in the `RETURN-CODE` special-register.

8.2.53. X"91"

X"91" Built-In Subroutine Syntax

```
CALL X"91" USING return-code, function-code, binary-variable-arg
~~~~       ~~~~~
```

The original Micro Focus version of this routine is capable of providing a wide variety of functions. GnuCOBOL supports just three of them:

- Turning runtime switches (`SWITCH-1`, . . . , `SWITCH-8`) on.
- Turning runtime switches (`SWITCH-1`, . . . , `SWITCH-8`) off.
- Retrieving the number of arguments passed to a subroutine.

The *return-code* argument must be a one-byte binary numeric data item (`USAGE BINARY-CHAR` is recommended). It will receive a value of 0 if the operation was successful, 1 otherwise.

The *function-code* argument must be either a numeric literal or a one-byte binary numeric data item (`USAGE BINARY-CHAR` is recommended).

The third argument — *variable-arg* — is defined differently depending upon the *function-code* value, as follows:

11 Sets and/or clears all eight of the COBOL switches (`SWITCH-1` through `SWITCH-8`). See [SPECIAL-NAMES], page 92, for an explanation of those switches.

The *variable-arg* argument should be an `OCCURS 8 TIMES` table of `USAGE BINARY-CHAR`.

Each occurrence that is set to a value of zero prior to the `CALL X"91"` will cause the corresponding switch to be cleared. Each occurrence set to 1 prior to the `CALL X"91"` will cause the corresponding switch to be set.

Values other than 0 or 1 will be ignored.

12 Reads all eight of the COBOL switches (`SWITCH-1` through `SWITCH-8`)

The *variable-arg* argument should be an `OCCURS 8 TIMES` table of `USAGE BINARY-CHAR`.

Each of the 1st eight occurrences of the array will be set to either 0 or 1 — 1 if the corresponding switch is set, 0 otherwise.

16 Retrieves the number of arguments passed to the program executing the `CALL X"91"`, saving that number into the *variable-arg* argument. That should be a binary numeric data item (`USAGE BINARY-CHAR` is recommended).

8.2.54. X"E4"

X"E4" Built-In Subroutine Syntax

```
CALL X"E4"
~~~~
```

Use `X"E4"` to clear the screen. There are no arguments and no returned value.

8.2.55. X"E5"

X"E5" Built-In Subroutine Syntax

```
CALL X"E5"
~~~~
```

The `X"E5"` routine will sound the PC "bell". There are no arguments and no returned value.

8.2.56. X"F4"

X"F4" Built-In Subroutine Syntax

```
CALL X"F4" USING byte, table
~~~~       ~~~~~
```

This routine packs the low-order (rightmost) bit from each of the eight 1-byte items in *table* into the corresponding bit positions of the single-byte data item *byte*.

The *byte* data item need be only a single byte in size. If it is longer, the excess will be unaffected by this subroutine.

The *table* data item must be at least 8 bytes long. If it is longer, the excess will be ignored by this subroutine.

Typically, table is defined similarly to the following:

```
01  Table-Arg.
    05 Each-Byte OCCURS 8 TIMES USAGE BINARY-CHAR.
```

8.2.57. X"F5"

X"F5" Built-In Subroutine Syntax

```
CALL X"F5" USING byte, table
~~~~       ~~~~~
```

This routine unpacks each bit of the single-byte data item *byte* into the low-order (rightmost) bit of each of the corresponding eight 1-byte items in *table*. The other seven bit positions of each of the first eight entries in *table* will be set to zero.

The *byte* data item need be only a single byte in size. If it is longer, the excess will be unaffected by this subroutine.

The *table* data item must be at least 8 bytes long. If it is longer, the excess will be ignored by this subroutine.

Typically, table is defined similarly to the following:

```
01  Table-Arg.
    05 Each-Byte OCCURS 8 TIMES USAGE BINARY-CHAR.
```

End of Chapter 8 — Functions

9. Report Writer Usage

9.1. RWCS Lexicon

There are a number of terms that describe various aspects of the operation of the Report Writer Control System (RWCS). Understanding the meanings of these terms is vital to developing an understanding of the subject.

Control Break
: An event that is triggered when a control field on an RWCS-generated report changes value. It is these events that trigger the generation of control heading and control footing groups.

Control Field
: A field of data being presented within a detail group; as the various detail groups that comprise the report are presented, they are presumed to appear in sorted sequence of the control fields contained within them. As an example, a department-by-department sales report for a chain of stores would probably be sorted by store number and – within like store numbers – be further sorted by department number. The store number will undoubtedly serve as a control field for the report, allowing control heading groups to be presented before each sequence of detail groups for the same store and control footing groups to be presented after each such sequence.

Control Footing
: A report group that appears immediately after one or more detail groups of an RWCS-generated report. Such are produced automatically as a result of a control break. This type of group typically serves as a summary of the detail group(s) that precede it, as might be the case on a sales report for a chain of stores, where the detail groups documenting sales for each department (one department per detail group) from the same store might be followed by a control footing that provides a summation of the department-by-department sales for that store.

Control Heading
: A report group that appears immediately before one or more detail groups of an RWCS-generated report. Such are produced automatically as a result of a control break. This type of group typically serves as an introduction to the detail group(s) that follow, as might be the case on a sales report for a chain of stores, where the detail groups documenting sales for each department (one department per detail group) from the same store might be preceded by a control heading that states the full name and location of the store.

Detail Group
: A report group that contains the detailed data being presented for the report.

Page Footing
: A report group that appears at the bottom of every page of an RWCS-generated report. Information typically found within such a report group might be:

- The date the report was generated
- The current page number of the report

Page Heading
: A report group that appears at the top of every page of an RWCS-generated report. Information typically found within such a report group might be:
  - A title for the report
  - The date the report was generated
  - The current page number of the report
  - Column headings describing the fields within the detail group(s)

Report Footing
: A report group that occurs only once in an RWCS-generated report — as the very last presented report group of the report. These typically serve as a visual indication that the report is finished.

Report Group
: One or more consecutive lines on a report that serve a common informational purpose or function. For example, lines of text that are displayed at the top or bottom of every printed page of a report.

Report Heading
: A report group that occurs only once in an RWCS-generated report — as the very first presented report group of the report. These typically serve as an introduction to the report.

9.2. The Anatomy of a Report

Every report has the same basic structure, as shown here, even though not all reports will have all of the groups shown. In fact, it is a very unusual report indeed that actually has every one of these groups:

- REPORT HEADING
- PAGE HEADING [1]
- CONTROL HEADING(S) [2]
- DETAIL GROUP(S) [2]
- CONTROL FOOTING(S) [2]
- FINAL CONTROL FOOTING
- PAGE FOOTING [1]
- REPORT FOOTING

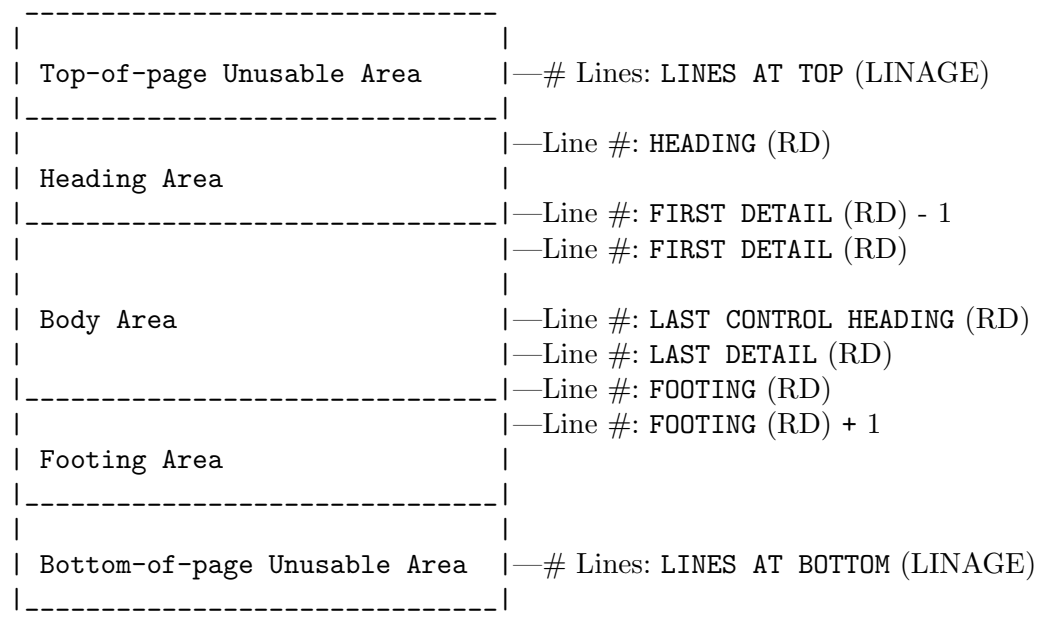
[1] Presented throughout the report, as needed
[2] Repeated, as needed

These groups will be presented (printed) across however many formatted pages are necessary to hold them. No single report group will be allowed to cross page boundaries.

The management of paging, enforcement of the *groups cannot span pages* rule and almost every aspect of report generation are handled entirely by the Report Writer Control System.

9.3. The Anatomy of a Report Page

Each page of a report is divided into as many as five (5) areas, as shown in the following diagram.



When describing a report via the `RD` (see [REPORT SECTION], page 135) clause, the total number of usable lines are specified as the `PAGE LIMIT` value; this value is the sum of the number of lines contained in the Heading, Body and Footing Areas.

The unusable areas of a page (if any) will appear above and below that usable area. You don't specify the unusable area in the `RD`, but rather using a `LINAGE` (see [File/Sort-Description], page 124) clause in the `FD` of the file the report is "attached" to.

The various report groups will be presentable in the various areas of a page, as follows:

`REPORT HEADING`
: Heading Area — An exception to this is the situation where the report heading report group contains the `NEXT GROUP NEXT PAGE` (see [NEXT GROUP], page 187) option; in those cases, the report heading will be presented on a page by itself (anywhere on that page) at the beginning of the report.

`PAGE HEADING`
: Heading Area

`CONTROL HEADING`
: Body Area, but no line of a control heading is allowed past the line number specified by `LAST CONTROL HEADING`

`DETAIL` Body Area, but no line of a detail report group is allowed past the line number specified by `LAST DETAIL`

`CONTROL FOOTING`
Body Area, but no line of a control footing report group is allowed past the line number specified by `FOOTING`

`PAGE FOOTING`
Footing Area

`REPORT FOOTING`
Footing Area — An exception to this is the situation where the report footing report group contains the `NEXT PAGE` option in its `LINE` (see [LINE], page 183) clause; in those cases, the report footing will be presented on a page by itself at the end of the report.

9.4. How RWCS Builds Report Pages

A report created via a `WRITE` statement (see [WRITE], page 417) will contain carriage-control information. Most notably, ASCII form-feed characters (X'0C') will be written to the report file to support the statement's `ADVANCING PAGE` option. Whether the data for a report line created via `ADVANCING PAGE` occurs *before* or *after* the form-feed character depends upon whether the programmer coded `WRITE` *`record-name`* `BEFORE ADVANCING PAGE` or `WRITE` *`record-name`* `AFTER ADVANCING PAGE`, respectively.

The GnuCOBOL implementation of RWCS does not issue any carriage-control information to the report files it produces — instead, it relies upon the information coded in the `RD` for the report (specifically the `PAGE LIMITS` and related options) and its internally-generated and managed `LINE-COUNTER` special register (see [Special Registers], page 265) for the report to know when to issue any blank lines to the file to fill-out the end of a printed page.

Because this is the way the GnuCOBOL RWCS works, in order to design an RWCS-generated report you'll need to know answers to the following questions:

1. What printer(s) will the report be printed on?
2. What paper orientation will you use, — Landscape (long edge of the paper at the top and bottom of page), or Portrait (long edge of the paper at the left and right of page)?
3. What tool will be used to print the report (direct printing to the device, notepad.exe, MS-Word, . . .)?
4. What font and font size will be used for the report when it is printed? RWCS-generated reports will assume that a fixed-width font such as "Courier", "Lucida Console", "Consolas" and the like will be used to print, as variable-pitch fonts would make the proper alignment of columns of data on reports virtually impossible.
5. When unprintable area exists at all four margins of the paper? These are generally caused by the printer itself or by its software driver.
6. What is the maximum number of lines per page that may be printed on a single sheet of paper?
7. What is the maximum number of characters that may be printed on one line?

Once you know the answer to questions 1-4, you may easily determine the answers to the remaining questions as follows:

1. Prepare a text file containing 100 or so records, each consisting of a numeric scale (`123456789012345678901234...`).
2. Print the file in a manner consistent with your answers to questions 1-4.
3. Add any necessary additional digits to each record in your test file (if lines weren't full) or remove characters from the end of each record if lines wrapped. If you made changes, reprint the file.
4. Now that you know *exactly* how long each record may be, add additional records and reprint. Continue until printing overflows to a second page.
5. The first page you print is now a perfect template to use when designing reports — it shows, given the answers to questions 1-4, every available printable character position on a page! The number of lines printed on that page becomes your `PAGE LIMIT` value for the `RD`.

The remaining `PAGE LIMIT` values can be established as required by your report(s).

Using *identifier* rather than *integer* specifications in the `RD` will give your program the ability — at run time — to accommodate multiple printers, fonts, font sizes and paper orientation. Just follow the above steps for each combination you wish your program to support.

9.5. Control Hierarchy

Every report that employs control breaks has a natural hierarchy of those control breaks based upon the manner in which the data the report is being generated from is sorted. This concept is best understood using an example which assumes a COBOL program to process sales data collected from every computerized cash register across a chain of stores having multiple departments is being developed.

The application that collects data from the various cash registers at each store will generate data records that look like this to a COBOL program:

```
01  Sales-For-Register.
    05 Sales-Date            PIC 9(8).
    05 Time-Collected        PIC 9(6).
    05 Register-Number       PIC 9(7).
    05 Store-Number          PIC 9(3).
    05 Department-Number     PIC 9(3).
    05 Total-Sales           PIC 9(6)V99.
```

Your task is to develop a report that shows the sales total from each cash register and summarizes those sales by department within each store, by store and also generates a total sales figure for the day across all stores.

To accomplish this, you will use a `SORT` statement (see [SORT], page 391) to sort the file of cash register sales data into:

1. Ascending sequence of store number
2. Within each store, data will be sorted into ascending sequence of department number
3. If there are multiple cash registers in a particular department of a specific store, the data needs to be further sorted so that the cash registers are ordered in sequence of their register number.

So, assuming a sort file has been defined and its record layout (essentially a mirror of the raw data file) is defined as follows:

```
01  Sorted-Sales-For-Register.
    05 Sorted-Sales-Date            PIC 9(8).
    05 Sorted-Time-Collected        PIC 9(6).
    05 Sorted-Register-Number       PIC 9(7).
    05 Sorted-Store-Number          PIC 9(3).
    05 Sorted-Department-Number     PIC 9(3).
    05 Sorted-Total-Sales           PIC 9(6)V99.
```

Then the `SORT` statement to accomplish the desired sequencing would be:

```
SORT SORT-FILE
    ASCENDING KEY Sorted-Store-Number
                  Sorted-Department-Number
                  Sorted-Register-Number
    USING Input-file
    OUTPUT PROCEDURE 100-Generate-Report
```

As a result of the sort, our program might expect to see data somewhat like this (date, time and sales totals are shown as ". . ."):

```
       +-------------------- Register Number
       |      +------------- Store Number
       |      |  +---------- Department Number
       |      |  |
...0535240001001...
...0589130001001...
...0625174001001...
...0122234001002...
...0732345001002...
...0003423001003...
...2038774001004...
...0112646002001...
...9963348002002...
...3245677002003...
...4456778002003...
...0002345002004...
```

Because of the sort, the most-frequently changing value of the three sort keys will be that of `Sorted-Register-Number`. This essentially defines the "detail" level of the report.

The next most-frequently changing value is that of `Sorted-Department-Number`, and the least-frequently changing value is that of `Sorted-Store-Number`. remember that the program should be generating totals each time one of these two values change, plus a grand total of sales at the end of the report. These three points are the *Control Break* points of the report.

When the report is defined, it's `RD` would contain a `CONTROLS ARE` clause that lists the control breaks in least- to most-frequent sequence of changing. This would be coded as:

```
CONTROLS ARE FINAL, Sorted-Store-Number, Sorted-Department-Number
```

A `FINAL` control break only occurs once, at the very end of the report. The `CONTROL FOOTING` for this break will be the one that produces the grand total of sales for all stores.

The next break listed on the `CONTROLS` clause will be the one that occurs next most-frequently (`Sorted-Store-Number`). This control break will be the one that produces the summation for each entire store, and will have its own `CONTROL FOOTING`.

The next (and last, in this case) break listed on the `CONTROLS` clause will be the one that occurs even more frequently (`Sorted-Department-Number`). The `CONTROL FOOTING` for this control field will be the one that summarizes sales for each department within a store.

This sequence of control breaks from least- to most-frequent (in other words, in the order they occur on the `CONTROLS ARE` clause) is the '*control hierarchy*' of the report; control breaks that occur more frequently than others are said to be at a lower level in the control hierarchy.

Defining a control hierarchy (via `CONTROLS ARE`) that does not match the actual sequence in which data will be processed is a great way to guarantee a "broken" report. I'll show you an example in a later section.

9.6. An Example

This section contains an example of the RWCS at work. The complete program, presented here, is a stripped-down version of a program I have used to generate a report for a class I teach on PC hardware. This report will provide benchmark statistics on a variety of popular AMD and Intel CPUs. The data for the report was obtained from the website `www.cpubenchmark.net` in December of 2013. By the time you are reading this, that data will most likely have become rather out of date, but it illustrates RWCS well enough.

9.6.1. Data

Here is the data that the program will be reading. Each record reflects the aggregated benchmark scoring for one particular CPU, as scores for benchmarks against that CPU have been reported to the `cpubenchmark.net` website by their PassMark benchmark software. The data consists of four fields. Fields are separated from one another by a single comma. The descriptions of the fields are as follows:

Benchmark Score
: A five-digit number showing the aggregated benchmark scores for the CPU; the higher this number, the better the CPU performed in benchmark testing.

Vendor
: The name of the vendor who makes the CPU. In this data, that will either be "AMD" (American Micro Devices) or "INTEL".

Family The 7-character family of CPU products the CPU falls into. This will have values such as "A4", "A10", "Core i5", "Core i7", etc.

Model The specific model of CPU within the family.

The first record of data shown below shows that the aggregated score of all benchmarks reported for the AMD A10-4600M CPU is 3145, as compared to the second record which shows that the aggregated score reported of all benchmarks reported for the Intel Core-i7-4960X CPU is 14291.

The following is the complete set of input data used for this example. This is by no means the complete set of data available at `cpubenchmark.net` — it is just a representative sample used for this example. For my class, I give my students a report showing the results for almost a thousand CPUs.

For the sake of brevity, this document lists the data in three columns.

```
03145,AMD,A10,4600M
14291,Intel,Core i7,4960X
02505,AMD,A10,4655M
03449,AMD,A10,4657M
04251,AMD,A10,5700
02758,AMD,A10,5745M
03332,AMD,A10,5750M
03253,AMD,A10,5757M
04798,AMD,A10,5800B
04677,AMD,A10,5800K
04767,AMD,A10,6700
05062,AMD,A10,6800K
00677,AMD,A4,1200
00559,AMD,A4,1250
01583,AMD,A4,3300
01237,AMD,A4,3300M
01227,AMD,A4,3305M
01263,AMD,A4,3310MX
01193,AMD,A4,3320M
01343,AMD,A4,3330MX
01625,AMD,A4,3400
01768,AMD,A4,3420
01685,AMD,A4,4300M
01169,AMD,A4,4355M
01919,AMD,A4,5000
01973,AMD,A4,5150M
02078,AMD,A4,5300
01632,AMD,A4,5300B
02305,AMD,A4,6300
01634,AMD,A6,1450
01964,AMD,A6,3400M
02101,AMD,A6,3410MX
02078,AMD,A6,3420M
02277,AMD,A6,3430MX
01995,AMD,A6,3500
02798,AMD,A6,3600
02892,AMD,A6,3620
03232,AMD,A6,3650
03327,AMD,A6,3670
01630,AMD,A6,4400M
05421,AMD,FX,6100
05813,AMD,FX,6120
06194,AMD,FX,6200
06388,AMD,FX,6300
07017,AMD,FX,6350
06163,AMD,FX,8100
06605,AMD,FX,8120
06845,AMD,FX,8140
07719,AMD,FX,8150
08131,AMD,FX,8320
09067,AMD,FX,8350
09807,AMD,FX,9370
10479,AMD,FX,9590
03076,Intel,Core i3,3110M
03301,Intel,Core i3,3120M
03655,Intel,Core i3,3130M
03820,Intel,Core i3,3210
02266,Intel,Core i3,3217U
04219,Intel,Core i3,3220
03724,Intel,Core i3,3220T
04407,Intel,Core i3,3225
02575,Intel,Core i3,3227U
01885,Intel,Core i3,3229Y
04259,Intel,Core i3,3240
03793,Intel,Core i3,3240T
04414,Intel,Core i3,3245
04757,Intel,Core i3,3250
03443,Intel,Core i3,4000M
02459,Intel,Core i3,4010U
02003,Intel,Core i3,4010Y
04904,Intel,Core i3,4130
04041,Intel,Core i3,4130T
05115,Intel,Core i3,4330
05117,Intel,Core i3,4340
03807,Intel,Core i5,3210M
03995,Intel,Core i5,3230M
03126,Intel,Core i5,3317U
04101,Intel,Core i5,3320M
05902,Intel,Core i5,3330
05690,Intel,Core i5,3330S
03917,Intel,Core i5,4300U
01743,Intel,Core i5,4300Y
04804,Intel,Core i5,4330M
03604,Intel,Core i5,4350U
06282,Intel,Core i5,4430
05954,Intel,Core i5,4430S
06517,Intel,Core i5,4440
07061,Intel,Core i5,4570
06474,Intel,Core i5,4570R
06803,Intel,Core i5,4570S
02503,Intel,Core i5,4570T
07492,Intel,Core i5,4670
07565,Intel,Core i5,4670K
06351,Intel,Core i5,4670T
03701,Intel,Core i7,3517U
03449,Intel,Core i7,3517UE
04588,Intel,Core i7,3520M
03912,Intel,Core i7,3537U
04861,Intel,Core i7,3540M
04009,Intel,Core i7,3555LE
06144,Intel,Core i7,3610QE
07532,Intel,Core i7,3610QM
06988,Intel,Core i7,3612QE
06907,Intel,Core i7,3612QM
05495,Intel,Core i7,3615QE
07310,Intel,Core i7,3615QM
07759,Intel,Core i7,3630QM
07055,Intel,Core i7,3632QM
06516,Intel,Core i7,3635QM
04032,Intel,Core i7,3667U
04271,Intel,Core i7,3687U
03479,Intel,Core i7,3689Y
08347,Intel,Core i7,3720QM
08512,Intel,Core i7,3740QM
09420,Intel,Core i7,3770
09578,Intel,Core i7,3770K
09074,Intel,Core i7,3770S
08280,Intel,Core i7,3770T
08995,Intel,Core i7,3820
08548,Intel,Core i7,3820QM
```

```
01296,AMD,A6,4455M
02440,AMD,A6,5200
01958,AMD,A6,5350M
01878,AMD,A6,5357M
01906,AMD,A6,5400B
02174,AMD,A6,5400K
02384,AMD,A6,6400K
02050,AMD,A8,3500M
02426,AMD,A8,3510MX
02245,AMD,A8,3520M
02276,AMD,A8,3530MX
02866,AMD,A8,3550MX
03215,AMD,A8,3800
03217,AMD,A8,3820
03552,AMD,A8,3850
03682,AMD,A8,3870K
02709,AMD,A8,4500M
02193,AMD,A8,4555M
04052,AMD,A8,5500
03464,AMD,A8,5500B
02434,AMD,A8,5545M
03052,AMD,A8,5550M
02935,AMD,A8,5557M
04348,AMD,A8,5600K
04390,AMD,A8,6500
04719,AMD,A8,6600K
04055,AMD,FX,4100
04153,AMD,FX,4130
04094,AMD,FX,4150
04774,AMD,FX,4170
04711,AMD,FX,4300
05247,AMD,FX,4350
05781,Intel,Core i5,3335S
03280,Intel,Core i5,3337U
02252,Intel,Core i5,3339Y
06282,Intel,Core i5,3340
04327,Intel,Core i5,3340M
05372,Intel,Core i5,3340S
06199,Intel,Core i5,3350P
04314,Intel,Core i5,3360M
04555,Intel,Core i5,3380M
03589,Intel,Core i5,3427U
03479,Intel,Core i5,3437U
03057,Intel,Core i5,3439Y
06442,Intel,Core i5,3450
06071,Intel,Core i5,3450S
06576,Intel,Core i5,3470
06077,Intel,Core i5,3470S
04591,Intel,Core i5,3470T
05991,Intel,Core i5,3475S
06828,Intel,Core i5,3550
06631,Intel,Core i5,3550S
06993,Intel,Core i5,3570
07118,Intel,Core i5,3570K
06709,Intel,Core i5,3570S
05414,Intel,Core i5,3570T
04333,Intel,Core i5,4200M
03355,Intel,Core i5,4200U
02358,Intel,Core i5,4200Y
02382,Intel,Core i5,4210Y
03482,Intel,Core i5,4250U
04381,Intel,Core i5,4258U
04663,Intel,Core i5,4288U
04786,Intel,Core i5,4300M
09025,Intel,Core i7,3840QM
09196,Intel,Core i7,3920XM
12107,Intel,Core i7,3930K
09052,Intel,Core i7,3940XM
12718,Intel,Core i7,3960X
12823,Intel,Core i7,3970X
03992,Intel,Core i7,4500U
04507,Intel,Core i7,4558U
04892,Intel,Core i7,4600M
04484,Intel,Core i7,4600U
03680,Intel,Core i7,4610Y
04345,Intel,Core i7,4650U
07352,Intel,Core i7,4700EQ
08161,Intel,Core i7,4700HQ
07946,Intel,Core i7,4700MQ
08002,Intel,Core i7,4702HQ
07647,Intel,Core i7,4702MQ
08066,Intel,Core i7,4750HQ
07367,Intel,Core i7,4765T
09969,Intel,Core i7,4770
10190,Intel,Core i7,4770K
09803,Intel,Core i7,4770S
08803,Intel,Core i7,4770T
10078,Intel,Core i7,4771
08567,Intel,Core i7,4800MQ
09969,Intel,Core i7,4820K
09331,Intel,Core i7,4850HQ
09323,Intel,Core i7,4900MQ
13620,Intel,Core i7,4930K
09754,Intel,Core i7,4930MX
10262,Intel,Core i7,4960HQ
```

9.6.2. Program

Here is the program that will be producing the report. Pay attention to how the data is sorted and how the control hierarchy (`CONTROLS ARE`) relates to the `SORT`.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DEMORWCS.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY. FUNCTION ALL INTRINSIC.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT CPU-FILE             ASSIGN TO "CPUDATA.txt"
                                LINE SEQUENTIAL.
    SELECT REPORT-FILE          ASSIGN TO "CPUREPORT.txt"
                                LINE SEQUENTIAL.
    SELECT SORT-FILE            ASSIGN TO DISK.
DATA DIVISION.
FILE SECTION.
FD  CPU-FILE.
```

```
01  CPU-REC                     PIC X(26).

FD  REPORT-FILE
    REPORT IS CPU-Report.

SD  SORT-FILE.
01  SORT-REC.
    05 F-SR-Score-NUM           PIC 9(5).
    05 F-SR-Vendor-TXT          PIC X(5).
    05 F-SR-Family-TXT          PIC X(7).
    05 F-SR-Model-TXT           PIC X(6).
WORKING-STORAGE SECTION.
01  WS-Date                     PIC 9(8).

01  WS-Family-Counters.
    05 WS-FC-AVE                PIC 9(5)V99.
    05 WS-FC-Qty                BINARY-LONG.
    05 WS-FC-Total-NUM          BINARY-LONG.

01  WS-Flags.
    05 WS-F-EOF                 PIC X(1).

01  WS-One-Const                PIC 9 VALUE 1.

01  WS-Overall-Counters.
    05 WS-OC-AVE                PIC 9(5)V99.
    05 WS-OC-Qty                BINARY-LONG.
    05 WS-OC-Total-NUM          BINARY-LONG.

01  WS-Starz                    PIC X(44) VALUE ALL '*'.

01  WS-Vendor-Counters.
    05 WS-VC-AVE                PIC 9(5)V99.
    05 WS-VC-Qty                BINARY-LONG.
    05 WS-VC-Total-NUM          BINARY-LONG.

REPORT SECTION.
RD  CPU-Report
    CONTROLS ARE FINAL
                 F-SR-Vendor-TXT
                 F-SR-Family-TXT

    PAGE LIMIT IS    36 LINES
        HEADING      1
        FIRST DETAIL 5
        LAST DETAIL  36.
```

```
01  TYPE IS PAGE HEADING.
    05 LINE NUMBER PLUS 1.
       10 COL 1  SOURCE WS-Date                PIC 9999/99/99.
       10 COL 14 VALUE 'CPU Benchmark Scores'.
       10 COL 37 VALUE 'Page:'.
       10 COL 43 SOURCE PAGE-COUNTER           PIC Z9.
    05 LINE NUMBER PLUS 1.
       10 COL 1  SOURCE WS-Starz               PIC X(44).
    05 LINE NUMBER PLUS 1.
       10 COL 1  VALUE '**'.
       10 COL 6  VALUE 'All CPU Data From cpubenchmark.net'.
       10 COL 43 VALUE '**'.
    05 LINE NUMBER PLUS 1.
       10 COL 1  SOURCE WS-Starz               PIC X(44).

01  TYPE CONTROL HEADING F-SR-Family-TXT.
    05 LINE NUMBER PLUS 1.
       10 COL 1  SOURCE F-SR-Vendor-TXT        PIC X(6).
       10 COL 8  SOURCE F-SR-Family-TXT        PIC X(7).
    05 LINE NUMBER PLUS 1.
       10 COL 1  VALUE 'Family'.
       10 COL 9  VALUE 'Model'.
       10 COL 16 VALUE 'Benchmark Score (High to Low)'.
    05 LINE NUMBER PLUS 1.
       10 COL 1  VALUE '======'.
       10 COL 9  VALUE '======'.
       10 COL 16 VALUE '============================'.

01  Detail-Line TYPE IS DETAIL.
    05 LINE NUMBER PLUS 1.
       10 COL 1  SOURCE F-SR-Family-TXT PIC X(7) GROUP INDICATE.
       10 COL 9  PIC X(6)       SOURCE F-SR-Model-TXT.
       10 COL 16 PIC ZZZZ9      SOURCE F-SR-Score-NUM.

01  End-Family TYPE IS CONTROL FOOTING F-SR-Family-TXT.
    05 LINE NUMBER PLUS 1.
       10 COL 9                 VALUE  'Ave...'.
       10 COL 16 PIC ZZZZ9.99   SOURCE WS-FC-AVE.
       10 COL 25                VALUE  '('.
       10 COL 26 PIC ZZ9        SUM    WS-One-Const.
       10 COL 30                VALUE  'Family CPUs)'.

01  End-Vendor TYPE IS CONTROL FOOTING F-SR-Vendor-TXT.
    05 LINE NUMBER PLUS 1.
       10 COL 9                 VALUE  'Ave...'.
```

```
           10 COL 16 PIC ZZZZ9.99   SOURCE WS-VC-AVE.
           10 COL 25                VALUE  '('.
           10 COL 26 PIC ZZ9        SUM    WS-One-Const.
           10 COL 30                VALUE  'Vendor CPUs)'.

       01  End-Overall TYPE IS CONTROL FOOTING FINAL.
           05 LINE NUMBER PLUS 1.
           10 COL 9                 VALUE  'Ave...'.
           10 COL 16 PIC ZZZZ9.99   SOURCE WS-OC-AVE.
           10 COL 25                VALUE  '('.
           10 COL 26 PIC ZZ9        SUM    WS-One-Const.
           10 COL 30                VALUE  'CPUs)'.

       PROCEDURE DIVISION.
       DECLARATIVES.
       000-End-Family SECTION.
           USE BEFORE REPORTING End-Family.
       1.  IF WS-FC-Qty > 0
               COMPUTE WS-FC-AVE = WS-FC-Total-NUM / WS-FC-Qty
           ELSE
               MOVE 0 TO WS-FC-AVE
           END-IF
           MOVE 0 TO WS-FC-Qty
                     WS-FC-Total-NUM
           .
       000-End-Vendor SECTION.
           USE BEFORE REPORTING End-Vendor.
       1.  IF WS-VC-Qty > 0
               COMPUTE WS-VC-AVE = WS-VC-Total-NUM / WS-VC-Qty
           ELSE
               MOVE 0 TO WS-VC-AVE
           END-IF
           MOVE 0 TO WS-VC-Qty
                     WS-VC-Total-NUM
           .
       000-End-Overall SECTION.
           USE BEFORE REPORTING End-Overall.
       1.  IF WS-OC-Qty > 0
               COMPUTE WS-OC-AVE = WS-OC-Total-NUM / WS-OC-Qty
           ELSE
               MOVE 0 TO WS-OC-AVE
           END-IF
           MOVE 0 TO WS-OC-Qty
                     WS-OC-Total-NUM
           .
       END DECLARATIVES.
```

```
010-Main SECTION.
1.  ACCEPT WS-Date FROM DATE YYYYMMDD
    SORT SORT-FILE
        ASCENDING KEY    F-SR-Vendor-TXT
                         F-SR-Family-TXT
        DESCENDING KEY   F-SR-Score-NUM
        ASCENDING KEY    F-SR-Model-TXT
        INPUT PROCEDURE  100-Pre-Process-Data
        OUTPUT PROCEDURE 200-Generate-Report
    STOP RUN
    .
100-Pre-Process-Data SECTION.
1.  OPEN INPUT CPU-FILE
    PERFORM FOREVER
        READ CPU-FILE
        AT END
            EXIT PERFORM
        END-READ
        MOVE SPACES TO SORT-REC
        UNSTRING CPU-REC DELIMITED BY ','
            INTO F-SR-Score-NUM,
                 F-SR-Vendor-TXT,
                 F-SR-Family-TXT,
                 F-SR-Model-TXT
        RELEASE SORT-REC
    END-PERFORM
    CLOSE CPU-FILE
    .
200-Generate-Report SECTION.
1.  INITIALIZE WS-Family-Counters
               WS-Flags
    OPEN OUTPUT REPORT-FILE
    INITIATE CPU-Report
    RETURN SORT-FILE
    AT END
        MOVE 'Y' TO WS-F-EOF
    END-RETURN
    PERFORM UNTIL WS-F-EOF = 'Y'
        GENERATE Detail-Line
        ADD 1              TO WS-FC-Qty
                              WS-OC-Qty
                              WS-VC-Qty
        ADD F-SR-Score-NUM TO WS-FC-Total-NUM
                              WS-OC-Total-NUM
                              WS-VC-Total-NUM
```

```
        RETURN SORT-FILE
        AT END
            MOVE 'Y' TO WS-F-EOF
        END-RETURN
    END-PERFORM
    TERMINATE CPU-Report
    CLOSE REPORT-FILE
    .
```

9.6.3. Generated Report Pages

Finally, here's the report the program generates!

```
2013/12/24   CPU Benchmark Scores   Page:  1
*****************************************
**   All CPU Data From cpubenchmark.net   **
*****************************************
AMD    A10
Family  Model  Benchmark Score (High to Low)
======  ====== ============================
A10     6800K   5062
        5800B   4798
        6700    4767
        5800K   4677
        5700    4251
        4657M   3449
        5750M   3332
        5757M   3253
        4600M   3145
        5745M   2758
        4655M   2505
        Ave...  3817.90 ( 11 Family CPUs)
AMD    A4
Family  Model  Benchmark Score (High to Low)
======  ====== ============================
A4      6300    2305
        5300    2078
        5150M   1973
        5000    1919
        3420    1768
        4300M   1685
        5300B   1632
        3400    1625
        3300    1583
        3330MX  1343
        3310MX  1263
        3300M   1237
```

```
        3305M   1227
        3320M   1193
-----------------------------------------

2013/12/24   CPU Benchmark Scores   Page:  2
*****************************************
**   All CPU Data From cpubenchmark.net   **
*****************************************
A4      4355M   1169
        1200     677
        1250     559
        Ave...  1484.47 ( 17 Family CPUs)
AMD    A6
Family  Model  Benchmark Score (High to Low)
======  ====== ============================
A6      3670    3327
        3650    3232
        3620    2892
        3600    2798
        5200    2440
        6400K   2384
        3430MX  2277
        5400K   2174
        3410MX  2101
        3420M   2078
        3500    1995
        3400M   1964
        5350M   1958
        5400B   1906
        5357M   1878
        1450    1634
        4400M   1630
        4455M   1296
        Ave...  2220.22 ( 18 Family CPUs)
AMD    A8
Family  Model  Benchmark Score (High to Low)
======  ====== ============================
A8      6600K   4719
        6500    4390
        5600K   4348
-----------------------------------------

2013/12/24   CPU Benchmark Scores   Page:  3
*****************************************
**   All CPU Data From cpubenchmark.net   **
*****************************************
```

```
A8      5500    4052
        3870K   3682
        3850    3552
        5500B   3464
        3820    3217
        3800    3215
        5550M   3052
        5557M   2935
        3550MX  2866
        4500M   2709
        5545M   2434
        3510MX  2426
        3530MX  2276
        3520M   2245
        4555M   2193
        3500M   2050
        Ave...  3148.68 ( 19 Family CPUs)
AMD    FX
Family  Model  Benchmark Score (High to Low)
======  ====== ============================
FX      9590   10479
        9370    9807
        8350    9067
        8320    8131
        8150    7719
        6350    7017
        8140    6845
        8120    6605
        6300    6388
        6200    6194
        8100    6163
        6120    5813
-----------------------------------------

2013/12/24   CPU Benchmark Scores   Page:  4
*****************************************
**   All CPU Data From cpubenchmark.net   **
*****************************************
FX      6100    5421
        4350    5247
        4170    4774
        4300    4711
        4130    4153
        4150    4094
        4100    4055
        Ave...  6457.00 ( 19 Family CPUs)
```

```
        Ave...  3448.86 ( 84 Vendor CPUs)
Intel  Core i3
Family  Model  Benchmark Score (High to Low)
======  ====== ============================
Core i3 4340    5117
        4330    5115
        4130    4904
        3250    4757
        3245    4414
        3225    4407
        3240    4259
        3220    4219
        4130T   4041
        3210    3820
        3240T   3793
        3220T   3724
        3130M   3655
        4000M   3443
        3120M   3301
        3110M   3076
        3227U   2575
        4010U   2459
        3217U   2266
        4010Y   2003
-----------------------------------------

2013/12/24   CPU Benchmark Scores   Page:  5
*****************************************
**   All CPU Data From cpubenchmark.net   **
*****************************************
Core i3 3229Y   1885
        Ave...  3677.76 ( 21 Family CPUs)
Intel  Core i5
Family  Model  Benchmark Score (High to Low)
======  ====== ============================
Core i5 4670K   7565
        4670    7492
        3570K   7118
        4570    7061
        3570    6993
        3550    6828
        4570S   6803
        3570S   6709
        3550S   6631
        3470    6576
        4440    6517
```

```
         4570R   6474
         3450    6442
         4670T   6351
         3340    6282
         4430    6282
         3350P   6199
         3470S   6077
         3450S   6071
         3475S   5991
         4430S   5954
         3330    5902
         3335S   5781
         3330S   5690
         3570T   5414
         3340S   5372
         4330M   4804
-----------------------------------------

2013/12/24   CPU Benchmark Scores   Page:  6
*****************************************
**   All CPU Data From cpubenchmark.net   **
*****************************************
Core i5 4300M   4786
        4288U   4663
        3470T   4591
        3380M   4555
        4258U   4381
        4200M   4333
        3340M   4327
        3360M   4314
        3320M   4101
        3230M   3995
        4300U   3917
        3210M   3807
        4350U   3604
        3427U   3589
        4250U   3482
        3437U   3479
        4200U   3355
        3337U   3280
        3317U   3126
        3439Y   3057
        4570T   2503
        4210Y   2382
        4200Y   2358
        3339Y   2252
```

```
        4300Y   1743
        Ave...  5026.13 ( 52 Family CPUs)
Intel  Core i7
Family  Model  Benchmark Score (High to Low)
======  ====== ============================
Core i7 4960X  14291
        4930K  13620
        3970X  12823
-----------------------------------------

2013/12/24   CPU Benchmark Scores   Page:  7
*****************************************
**   All CPU Data From cpubenchmark.net   **
*****************************************
Core i7 3960X  12718
        3930K  12107
        4960HQ 10262
        4770K  10190
        4771   10078
        4770    9969
        4820K   9969
        4770S   9803
        4930MX  9754
        3770K   9578
        3770    9420
        4850HQ  9331
        4900MQ  9323
        3920XM  9196
        3770S   9074
        3940XM  9052
        3840QM  9025
        3820    8995
        4770T   8803
        4800MQ  8567
        3820QM  8548
        3740QM  8512
        3720QM  8347
        3770T   8280
        4700HQ  8161
        4750HQ  8066
        4702HQ  8002
        4700MQ  7946
        3630QM  7759
        4702MQ  7647
        3610QM  7532
        4765T   7367
```

```
-----------------------------------------

2013/12/24   CPU Benchmark Scores   Page:  8
*****************************************
**   All CPU Data From cpubenchmark.net   **
*****************************************
Core i7 4700EQ  7352
        3615QM  7310
        3632QM  7055
        3612QE  6988
        3612QM  6907
        3635QM  6516
        3610QE  6144
        3615QE  5495
        4600M   4892
        3540M   4861
        3520M   4588
        4558U   4507
        4600U   4484
        4650U   4345
        3687U   4271
        3667U   4032
        3555LE  4009
        4500U   3992
        3537U   3912
        3517U   3701
        4610Y   3680
        3689Y   3479
        3517UE  3449
        Ave...  7725.58 ( 58 Family CPUs)
        Ave...  6005.16 (131 Vendor CPUs)
        Ave...  5006.42 (215 CPUs)




-----------------------------------------
```

9.7. Control Hierarchy (Revisited)

The sample program just discussed presents a great opportunity to show what can happen if you don't define the control hierarchy of a report properly.

I changed the `CONTROLS ARE` clause on the sample program from this:

```
CONTROLS ARE FINAL
             F-SR-Vendor-TXT
```

```
                F-SR-Family-TXT
```

To this:

```
CONTROLS ARE FINAL
             F-SR-Family-TXT
             F-SR-Vendor-TXT
```

And then ran the report again. Here are the first two pages of that new report. See what happened to the control breaks?

```
2013/12/24   CPU Benchmark Scores   Page:  1
*******************************************
**   All CPU Data From cpubenchmark.net   **
*******************************************
AMD    A10
Family  Model  Benchmark Score (High to Low)
======  ====== ============================
A10     6800K   5062
        5800B   4798
        6700    4767
        5800K   4677
        5700    4251
        4657M   3449
        5750M   3332
        5757M   3253
        4600M   3145
        5745M   2758
        4655M   2505
        Ave...  3817.90 ( 11 Vendor CPUs)
        Ave...  3817.90 ( 11 Family CPUs)
AMD    A4
Family  Model  Benchmark Score (High to Low)
======  ====== ============================
A4      6300    2305
        5300    2078
        5150M   1973
        5000    1919
        3420    1768
        4300M   1685
        5300B   1632
        3400    1625
        3300    1583
        3330MX  1343
        3310MX  1263
        3300M   1237
        3305M   1227
```

```
-----------------------------------------

2013/12/24   CPU Benchmark Scores   Page:  2
*****************************************
**   All CPU Data From cpubenchmark.net   **
*****************************************
A4      3320M   1193
        4355M   1169
        1200     677
        1250     559
        Ave...  1484.47 ( 17 Vendor CPUs)
        Ave...  1484.47 ( 17 Family CPUs)
AMD    A6
Family  Model  Benchmark Score (High to Low)
======  ====== ============================
A6      3670    3327
        3650    3232
        3620    2892
        3600    2798
        5200    2440
        6400K   2384
        3430MX  2277
        5400K   2174
        3410MX  2101
        3420M   2078
        3500    1995
        3400M   1964
        5350M   1958
        5400B   1906
        5357M   1878
        1450    1634
        4400M   1630
        4455M   1296
        Ave...  2220.22 ( 18 Vendor CPUs)
        Ave...  2220.22 ( 18 Family CPUs)
AMD    A8
Family  Model  Benchmark Score (High to Low)
======  ====== ============================
A8      6600K   4719

-----------------------------------------
```

9.8. Turning PHYSICAL Page Formatting Into LOGICAL Formatting

You can trick RWCS into using the `PAGE LIMIT` values as logical specifications rather than physical ones quite easily — simply include an ASCII form-feed (`X'0C'`) character into your page heading design! Here's how the sample program shown earlier could be easily modified:

Simply Change This. . .

```
01  TYPE IS PAGE HEADING.
    05 LINE NUMBER 1.
       10 COL 1  SOURCE WS-Date PIC 9999/99/99.
       10 COL 14 VALUE 'CPU Benchmark Scores'.
       10 COL 37 VALUE 'Page:'.
       10 COL 43 SOURCE PAGE-COUNTER PIC Z9.
    05 LINE NUMBER PLUS 1.
       10 COL 1  SOURCE WS-Starz PIC X(44).
    05 LINE NUMBER PLUS 1.
       10 COL 1  VALUE '**'.
       10 COL 6  VALUE 'All CPU Data From ' &
                       'cpubenchmark.net'.
       10 COL 43 VALUE '**'.
    05 LINE NUMBER PLUS 1.
       10 COL 1  SOURCE WS-Starz PIC X(44).
```

To This. . .

```
01  TYPE IS PAGE HEADING.
  05 LINE NUMBER 1.                  *> NEW
    10 COL 1  VALUE X'0C'.           *> NEW
  05 LINE NUMBER PLUS 1.             *> CHANGED
       10 COL 1  SOURCE WS-Date PIC 9999/99/99.
       10 COL 14 VALUE 'CPU Benchmark Scores'.
       10 COL 37 VALUE 'Page:'.
       10 COL 43 SOURCE PAGE-COUNTER PIC Z9.
    05 LINE NUMBER PLUS 1.
       10 COL 1  SOURCE WS-Starz PIC X(44).
    05 LINE NUMBER PLUS 1.
       10 COL 1  VALUE '**'.
       10 COL 6  VALUE 'All CPU Data From ' &
                       'cpubenchmark.net'.
       10 COL 43 VALUE '**'.
    05 LINE NUMBER PLUS 1.
       10 COL 1  SOURCE WS-Starz PIC X(44).
```

RWCS will still be counting lines to decide when to close off one page and start a new one, but when a new page is started its page heading will *physically* form-feed the printer when the report is printed. As long as any printer you plan on using supports at least as

many physical print lines as what is defined as the `PAGE LIMIT` value in whatever paper orientation and font you plan on (or are limited to) printing in, you have now divorced your program from the physical realities of the printer!

Of course, whatever software you are using to deliver the printed document to the printer with, must allow the ASCII form-feed character to pass through to the printer.

End of Chapter 9 — Report Writer Usage

10. Interfacing With The OS

10.1. Compiling Programs

Program source files should have by convention, extensions of `.cob` or `.cbl`.

Program file names should match exactly the specification of `PROGRAM-ID` (including case).

Spaces cannot be included in primary entry-point names and therefore must not be included in program file names.

The GnuCOBOL compiler will translate your COBOL program into C source code, compile that C source code into executable binary form using the C compiler specified when GnuCOBOL was built and link that executable binary into:

Directly executable form
: This is an executable file directly-executable from the command-line. On Windows computers, this would be an `.exe` file. On Unix systems, this will be a file with no specific extension, but with execute permissions. This file will include the main program as well as any static-linked subprograms.

Static-linkable form
: This is a single subprogram compiled into object-code form, ready to be linked in with a main program to form a directly-executable program. On windows computers, these generally are `.o` (object-code) files.

Dynamically-loadable executable form
: These are dynamically-loadable object code files ready to be invoked from other programs at execution time. On Windows systems, these would be `.dll` files, while on Unix systems they are typically `.so` files (OSX uses `.dylib`).

10.1.1. cobc - The GnuCOBOL Compiler - Runtime options

The GnuCOBOL compiler is named `cobc` (`cobc.exe` on a Windows system).

The following describes the syntax and option switches of the cobc command. This information may be displayed by entering the command `cobc --help` or `cobc -h`.

```
GnuCOBOL compiler for most COBOL dialects with lots of extensions

Sorted within groups

Usage: cobc [options]... file...

Options:
  -h, --help             display this help and exit

  -du(mpversion)         display compiler version and exit
  -i, --info             display compiler information (build/environment)
                         and exit
```

```
-q, --brief             reduced displays, commands invoked not shown
-v, --verbose           verbose mode, display additional information;
                        multiple -v options increase the verbosity,
                        the maximum is 3 as follows:
                        (1) display compiler version and the commands
                        invoked by the compiler,
                        (2) pass verbose option to assembler/compiler
                        (3) pass verbose option to linker
-V, --version           display compiler version information and exit
-###                    like -v but commands not executed

-A <options>            add <options> to the C compile phase
-b                      combine all input files into a single
                        dynamically loadable module
-c                      compile and assemble, but do not link
--conf=<file>           user-defined dialect configuration; see -std
-C                      translation only; convert COBOL to C
-d, --debug             enable all run-time error checking,
                        equal to -fstack-check -fec=EC-ALL
-D <define>             define <define> for COBOL compilation
-ext <extension>        add file extension for resolving COPY
-E                      preprocess only; do not compile or link
-F, --free              use free source format (alias for -fformat=free)
--fixed                 use fixed source format (default; alias for
                        -fformat=fixed)
-fec=<exception-name>   enable code generation for <exception-name>,
                        see --list-exceptions for the possible values,
                        sets -fsource-location
-fno-ec=<exception-name>
                        disable code generation for <exception-name>
-g                      enable C compiler debug and stack check
-I <directory>          add <directory> to copy/include search path
-j [<args>], --job[=<args>]
                        run program after build, passing <args>
-K <entry>              generate CALL to <entry> as static
-l <lib>                link the library <lib>
-L <directory>          add <directory> to library search path
--list-exceptions       display exception names
--list-intrinsics       display intrinsic functions
--list-mnemonics        display mnemonic names
--list-reserved         display reserved words
--list-system           display system routines
-m                      build a dynamically loadable module (default)
-o <file>               place the output into <file>
-O, -O2, -O3, -Os       enable optimization
-O0                     disable optimization
```

```
  -P[=<dir or file>]     generate preprocessed program listing (.lst)
  -Q <options>           add <options> to the C link phase
  -std=<dialect>         warnings/features for a specific dialect
                         <dialect> can be one of:
                         default, cobol2014, cobol2002, cobol85, xopen,
                         ibm-strict, ibm, mvs-strict, mvs,
                         mf-strict, mf, bs2000-strict, bs2000,
                         acu-strict, acu, rm-strict, rm, gcos-strict,
                         gcos;
                         see configuration files in directory config
  -S                     compile only; output assembly file
  --save-temps[=<dir>]   save intermediate files
                         * default: current directory
  -t <file>              generate and place a program listing into <file>
  --tlines=<lines>       specify lines per page in listing, default = 55
  -T <file>              generate and place a wide program listing into <file>
  -x                     build an executable program
  -X, --Xref             generate internal cross reference


Warning options:
  -w                     disable all warnings
  -Wall                  enable most warnings (all except as noted below)
  -Wadditional           additional warnings only raised with -Wall
  -Warchaic              warn if archaic features are used
  -Warithmetic-osvs      warn if arithmetic expression precision has changed
  -Wcall-params          warn about non 01/77 items for CALL parameters
                         * NOT set with -Wall
  -Wcolumn-overflow      warn about text after program-text area, FIXED format
                         * NOT set with -Wall
  -Wconstant-expression  warn about expressions that always resolve to true/false
  -Wconstant-numlit-expression
                         warn about numeric expressions that always resolve to
                         true/false
  -Wdangling-text        warn about source text after program-area
                         * NOT set with -Wall
  -Werror                treat all warnings as errors
  -Werror=<warning>      treat specified <warning> as error
  -Wextra                like -Wall but enable some extra warning flags
  -Wimplicit-define      warn whenever data items are implicitly defined
                         * NOT set with -Wall
  -Winitial-value        warn if initial VALUE clause is ignored
  -Wlarger-01-redefines  warn about larger redefines allowed by COBOL standards
  -Wlinkage              warn about dangling LINKAGE items
                         * NOT set with -Wall
  -Wobsolete             warn if obsolete features are used
```

```
-Woverlap               warn about overlapping MOVE of items
-Wno-corresponding      do not warn about CORRESPONDING with no matching items
                        * ALWAYS active
-Wno-dialect            do not warn about dialect specific issues
                        * ALWAYS active
-Wgoto-different-section
                        warn about GO TO a praragraph defined in a different section
-Wno-error              don't treat warnings as errors
-Wno-error=<warning>    don't treat specified <warning> as error
-Wno-goto-section       do not warn about GO TO section-name
                        * ALWAYS active
-Wno-ignored-error      do not warn about errors in code parts which are
                        unreachable and so normally ignored
                        * ALWAYS active
-Wno-missing-newline    do not warn about missing newlines
                        * ALWAYS active
-Wno-others             do not warn about different issues
                        * ALWAYS active
-Wno-pending            do not warn if pending features are used
                        * ALWAYS active
-Wno-repository-checks
                        do not warn/check for program/function/external signature
                        mismatch
                        * ALWAYS active
-Wno-unfinished         do not warn if unfinished features are used
                        * ALWAYS active
-Wno-unsupported        do not warn if runtime does not support a feature used
-Wno-<warning>          disable warning enabled by default, -Wall or -Wextra
-Wparentheses           warn if parentheses are omitted around AND within OR
-Wpossible-overlap      warn about MOVE of items that may overlap depending on
                        variables
                        * NOT set with -Wall
-Wpossible-truncate     warn about possible field truncation
                        * NOT set with -Wall
-Wprototypes            warn about missing FUNCTION prototypes/definitions
-Wredefinition          warn about non-referenced ambiguous data items
-Wstrict-typing         warn strictly about type mismatch
-Wsuspicious-perform-thru
                        warn if PERFORM THRU references procedures not in ascending
                        order or multiple sections
                        * ALWAYS active
-Wterminator            warn about lack of scope terminator END-XXX
                        * NOT set with -Wall
-Wtruncate              warn about field truncation from constant assignments
-Wunreachable           warn about likely unreachable statements
                        * NOT set with -Wall
```

```
Compiler options:
  -fsign=[ASCII|EBCDIC]  define display sign representation
                         * default: machine native
  -ffold-copy=[UPPER|LOWER]
                         fold COPY subject to value
                         * default: no transformation
  -ffold-call=[UPPER|LOWER]
                         fold PROGRAM-ID, CALL, CANCEL subject to value
                         * default: no transformation
  -fmax-errors=<number>  maximum number of errors to report before
                         compilation is aborted
                         * default: 128
  -fintrinsics=[ALL|intrinsic
                         function name(,name,...)]
                         intrinsics to be used without FUNCTION keyword
  -fdump=<scope>         dump data fields on abort, <scope> may be
                         a combination of: ALL, WS, LS, RD, FD, SC, LO
  -fcallfh=<name>        specifies <name> to be used for I/O
                         as external provided EXTFH interface module
  -febcdic-table=[DEFAULT|RESTRICTED-GC|IBM|GCOS]
                         define EBCDIC translation table:
                         * default: translation to extended ASCII as per MF
                         * restricted-gc: translation from restricted ASCII only
                         * ibm: translation to restricted ASCII as per IBM
                         * gcos: translation to extended ASCII as per GCOS7
  -fdefault-colseq=[ASCII|EBCDIC|NATIVE]
                         define default collating sequence
                         * default: NATIVE
  -fstack-extended       store origin of entrypoints and PERFORM
                         * turned on by -debug/-dump
  -fno-remove-unreachable
                         disable remove of unreachable code
                         * turned off by -g
  -ftrace                generate trace code
                         * scope: executed SECTION/PARAGRAPH
  -ftraceall             generate trace code
                         * scope: executed SECTION/PARAGRAPH/STATEMENTS
  -fsyntax-only          syntax error checking only; don't emit any output
  -fdebugging-line       enable debugging lines
                         * 'D' in indicator column or floating >>D
  -fsource-location      generate source location code
                         * turned on by -debug/-ftraceall/-fec/-dump
  -fimplicit-init        automatic initialization of the COBOL runtime system
  -fno-recursive-check   disable check of recursive program call;
```

```
                        effectively compiling as RECURSIVE program
  -fstack-check         PERFORM stack checking
                        * turned on by -debug/-g
  -fsection-exit-check  check that code execution does not leave the scope of
                        SECTIONs
  -fimplicit-goback-check
                        check that code execution does not end implicit at end of
                        PROCEDURE DIVISION
  -fwrite-after         use AFTER 1 for WRITE of LINE SEQUENTIAL
                        * default: BEFORE 1
  -fmfcomment           '*' in column 1 treated as comment with listing suppression
                        * FIXED/COBOL85/VARIABLE format only
  -facucomment          '$' in indicator area treated as '*',
                        '|' treated as floating comment
  -fno-trunc            allow numeric field overflow
                        * non-ANSI behaviour
  -fsingle-quote        use a single quote (apostrophe) for QUOTE
                        * default: double quote
  -foptional-file       treat all files as OPTIONAL
                        * unless NOT OPTIONAL specified
  -fstatic-call         output static function calls for the CALL statement
  -fno-gen-c-decl-static-call
                        disable generation of C function declarations
                        for subroutines with static CALL
  -fgen-c-line-directives
                        generate source location directives in C code;
                        * turned on by -g
  -fgen-c-labels        generate extra labels in C sources;
                        * turned on by -g
  -fno-theaders         suppress all headers and output of compilation
                        options from listing while keeping page breaks
  -fno-tsource          suppress source from listing
  -fno-tmessages        suppress warning and error summary from listing
  -ftsymbols            specify symbols in listing
  -fno-diagnostics-show-option
                        suppress output of option that directly
                        controls the diagnostic


Compiler dialect configuration options:
  -freserved-words=<value>
                        use of complete/fixed reserved words
  -ftab-width=1..12     number of spaces that are assumed for tabs
  -ftext-column=72..255 right margin column number for fixed-form reference format
  -fpic-length=<number> maximum number of characters allowed in the PICTURE
                        character-string
```

```
-fword-length=1..63    maximum word-length for COBOL (= programmer defined) words
-fliteral-length=<number>
                       maximum literal size in general
-fnumeric-literal-length=1..38
                       maximum numeric literal size
-fdefaultbyte=<value>  default initialization for fields without VALUE, may be
                       one of
                       * character in quotes
                       * decimal 0..255 representing a character
                       * "init" to initialize to PICTURE/USAGE
                       * "none" to do no explicit initialization
                       * default: "init"
-fformat=<value>       default reference-format, may be one of: FIXED, FREE,
                       COBOL85, VARIABLE, XOPEN, XCARD, CRT, TERMINAL, COBOLX
-fbinary-size=<value>  binary byte size - defines the allocated bytes according to
                       PIC, may be one of: 2-4-8, 1-2-4-8, 1--8
-fbinary-byteorder=<value>
                       binary byte order, may be one of: native, big-endian
-fassign-clause=<value>
                       how to interpret 'ASSIGN word': as 'ASSIGN EXTERNAL word' or
                       'ASSIGN DYNAMIC word', may be one of: dynamic, external,
                       ibm (= external), mf (= dynamic)
-fscreen-section-rules=<value>
                       which compiler's rules to apply to SCREEN SECTION item
                       clauses, may be one of: acu, gc, mf, rm, std, xopen
-fdpc-in-data=<value>  whether DECIMAL-POINT IS COMMA has effect in XML/JSON
                       GENERATE, may be one of: none, xml, json, all
-ffilename-mapping     resolve file names at run time using environment variables
-fpretty-display       alternate formatting of numeric fields
-fbinary-truncate      numeric truncation according to ANSI
-fcomplex-odo          allow complex OCCURS DEPENDING ON
-fodoslide             adjust items following OCCURS DEPENDING (implies complex-odo)
-findirect-redefines   allow REDEFINES to other than last equal level number
-frelax-syntax-checks  allow certain syntax variations (e.g. REDEFINES position)
-fref-mod-zero-length  allow zero length reference-modification (only changed with
                       EC-BOUND-REF-MOD active)
-frelax-level-hierarchy
                       allow non-matching level numbers
-fselect-working       require ASSIGN USING items to be in WORKING-STORAGE
-flocal-implies-recursive
                       LOCAL-STORAGE SECTION implies RECURSIVE attribute
-fsticky-linkage       LINKAGE SECTION items remain allocated between invocations
-fmove-ibm             MOVE operates as on IBM (left to right, byte by byte)
-fperform-osvs         exit point of any currently executing perform is recognized
                       if reached
-farithmetic-osvs      limit precision in intermediate results to precision of final
```

```
                        result (less accurate)
  -fconstant-folding    evaluate constant expressions at compile time
  -fhostsign            allow hexadecimal value 'F' for NUMERIC test of signed
                        PACKED DECIMAL field
  -fprogram-name-redefinition
                        program names don't lead to a reserved identifier
  -faccept-update       set WITH UPDATE clause as default for ACCEPT dest-item,
                        instead of WITH NO UPDATE
  -faccept-auto         set WITH AUTO clause as default for ACCEPT dest-item,
                        instead of WITH TAB
  -fconsole-is-crt      assume CONSOLE IS CRT if not set otherwise
  -fno-echo-means-secure        NO-ECHO
                        hides input with asterisks like SECURE
  -fline-col-zero-default
                        assume a field DISPLAY starts at LINE 0 COL 0 (i.e. at the
                        cursor), not LINE 1 COL 1
  -fdisplay-special-fig-consts
                        special behaviour of DISPLAY SPACE/ALL X'01'/ALL X'02'/ALL
                        X'07'
  -fbinary-comp-1       COMP-1 is a 16-bit signed integer
  -fnumeric-pointer     POINTER is a 64-bit unsigned integer
  -fmove-non-numeric-lit-to-numeric-is-zero
                        imply zero in move of non-numeric literal to numeric items
  -fimplicit-assign-dynamic-var
                        implicitly define a variable if an ASSIGN DYNAMIC does not
                        match any data item
  -fdevice-mnemonics    specifying device by mnemonic
  -fxml-parse-xmlss     XML PARSE XMLSS
  -fareacheck           check contents of Area A (when reference format supports
                        Area A enforcement),
                        enabled checks include:
                        * division, section, paragraph names, level indicators (FD,
                        SD, RD, and CD),
                          and toplevel numbers (01 and 77) must start in Area A;
                        * statements must not start in Area A; and
                        * separator periods must not be within Area A
  -fcomment-paragraphs=<support>
                        comment paragraphs in IDENTIFICATION DIVISION (AUTHOR,
                        DATE-WRITTEN, ...)
  -fcontrol-division=<support>
                        CONTROL DIVISION
  -fpartial-replace-when-literal-src=<support>
                        apply partial replacing with literal source operand even
                        when it replaces with spaces only;
                        * "skip" prevents such replacements
  -fmemory-size-clause=<support>
```

```
                        MEMORY-SIZE clause
  -fmultiple-file-tape-clause=<support>
                        MULTIPLE-FILE-TAPE clause
  -flabel-records-clause=<support>
                        LABEL-RECORDS clause
  -fvalue-of-clause=<support>
                        VALUE-OF clause
  -fdata-records-clause=<support>
                        DATA-RECORDS clause
  -ftop-level-occurs-clause=<support>
                        OCCURS clause on top-level
  -fsame-as-clause=<support>
                        SAME AS clause
  -ftype-to-clause=<support>
                        TYPE TO clause
  -fusage-type=<support>        USAGE type-name
  -fsynchronized-clause=<support>
                        SYNCHRONIZED clause
  -fsync-left-right=<support>
                        LEFT/RIGHT phrases in SYNCHRONIZED clause
  -fspecial-names-clause=<support>
                        SPECIAL-NAMES clause
  -fgoto-statement-without-name=<support>
                        GOTO statement without name
  -fstop-literal-statement=<support>
                        STOP-literal statement
  -fstop-identifier-statement=<support>
                        STOP-identifier statement
  -fstop-error-statement=<support>
                        STOP ERROR statement
  -fdebugging-mode=<support>
                        DEBUGGING MODE and debugging indicator
  -fuse-for-debugging=<support>
                        USE FOR DEBUGGING
  -fpadding-character-clause=<support>
                        PADDING CHARACTER clause
  -fnext-sentence-phrase=<support>
                        NEXT SENTENCE phrase
  -flisting-statements=<support>
                        listing-directive statements EJECT, SKIP1, SKIP2, SKIP3
  -ftitle-statement=<support>
                        listing-directive statement TITLE
  -fentry-statement=<support>
                        ENTRY statement
  -fmove-noninteger-to-alphanumeric=<support>
                        move noninteger to alphanumeric
```

```
-fmove-figurative-constant-to-numeric=<support>
                        move figurative constants to numeric
-fmove-figurative-space-to-numeric=<support>
                        move figurative constant SPACE to numeric
-fmove-figurative-quote-to-numeric=<support>
                        move figurative constant QUOTE to numeric
-fodo-without-to=<support>
                        OCCURS DEPENDING ON without to
-fsection-segments=<support>
                        section segments
-falter-statement=<support>
                        ALTER statement
-fcall-overflow=<support>
                        OVERFLOW clause for CALL
-fnumeric-boolean=<support>
                        boolean literals (B'1010')
-fhexadecimal-boolean=<support>
                        hexadecimal-boolean literals (BX'A')
-fnational-literals=<support>
                        national literals (N'UTF-16 string')
-fhexadecimal-national-literals=<support>
                        hexadecimal-national literals (NX'265E')
-fnational-character-literals=<support>
                        non-standard national literals (NC'UTF-16 string')
-fhp-octal-literals=<support>
                        HP COBOL octal literals (%377)
-facu-literals=<support>
                        ACUCOBOL-GT literals (#B #O #H #X)
-fword-continuation=<support>
                        continuation of COBOL words
-fnot-exception-before-exception=<support>
                        NOT ON EXCEPTION before ON EXCEPTION
-faccept-display-extensions=<support>
                        extensions to ACCEPT and DISPLAY
-frenames-uncommon-levels=<support>
                        RENAMES of 01-, 66- and 77-level items
-flarger-redefines=<support>
                        allow larger REDEFINES items
-fsymbolic-constant=<support>
                        constants defined in SPECIAL-NAMES
-fconstant-78=<support>
                        constant with level 78 item (note: has left to right
                        precedence in expressions)
-fconstant-01=<support>
                        constant with level 01 CONSTANT AS/FROM item
-fperform-varying-without-by=<support>
```

```
                        PERFORM VARYING without BY phrase (implies BY 1)
  -freference-out-of-declaratives=<support>
                        references to sections not in DECLARATIVES from within
                        DECLARATIVES
  -fprogram-prototypes=<support>
                        CALL/CANCEL with program-prototype-name
  -fcall-convention-mnemonic=<support>
                        specifying call-convention by mnemonic
  -fcall-convention-linkage=<support>
                        specifying call-convention by WITH ... LINKAGE
  -fnumeric-value-for-edited-item=<support>
                        numeric literals in VALUE clause of numeric-edited items
  -fincorrect-conf-sec-order=<support>
                        incorrect order of CONFIGURATION SECTION paragraphs
  -fdefine-constant-directive=<support>
                        allow >> DEFINE CONSTANT var AS literal
  -ffree-redefines-position=<support>
                        REDEFINES clause not following entry-name in definition
  -frecords-mismatch-record-clause=<support>
                        record sizes does not match RECORD clause
  -frecord-delimiter=<support>
                        RECORD DELIMITER clause
  -fsequential-delimiters=<support>
                        BINARY-SEQUENTIAL and LINE-SEQUENTIAL phrases in
                        RECORD DELIMITER
  -frecord-delim-with-fixed-recs=<support>
                        RECORD DELIMITER clause on file with fixed-length records
  -fmissing-statement=<support>
                        missing statement (e.g. empty IF / PERFORM)
  -fmissing-period=<support>
                        missing period in PROCEDURE DIVISION (when reference format
                        supports Area A enforcement)
  -fzero-length-literals=<support>
                        zero-length literals, e.g. '' and ""
  -fxml-generate-extra-phrases=<support>
                        XML GENERATE's phrases other than COUNT IN
  -fcontinue-after=<support>
                        AFTER phrase in CONTINUE statement
  -fgoto-entry=<support>        ENTRY FOR GOTO and GOTO ENTRY statements
  -fassign-variable=<support>
                        ASSIGN [TO] variable in SELECT
  -fassign-using-variable=<support>
                        ASSIGN USING/VARYING variable in SELECT
  -fassign-ext-dyn=<support>
                        ASSIGN EXTERNAL/DYNAMIC in SELECT
  -fassign-disk-from=<support>
```

```
                        ASSIGN DISK FROM variable in SELECT
 -fvsam-status=<support>
                        VSAM status in FILE STATUS
 -fself-call-recursive=<support>
                        CALL to own PROGRAM-ID implies RECURSIVE attribute
 -frecord-contains-depending-clause=<support>
                        DEPENDING clause in RECORD CONTAINS
 -fpicture-l=<support>  PICTURE string with 'L' character
       where <support> is one of the following:
       'ok', 'warning', 'archaic', 'obsolete', 'skip', 'ignore', 'error',
       'unconformable'

       -fnot-reserved=<word>   word to be taken out of the reserved words list
-freserved=<word>       word to be added to reserved words list
-freserved=<word>:<alias>
                        word to be added to reserved words list as alias
-fnot-register=<word>   special register to disable
-fregister=<word> or <word>:<definition>
                        special register to enable
```

Each file specified on the `cobc` command constitutes a *Compilation Unit*. A compilation unit may be a single GnuCOBOL program — with or without nested subprograms(see [Independent vs Contained vs Nested Subprograms], page 679) — or multiple GnuCOBOL programs, separated by `END PROGRAM` or `END FUNCTION` marker lines, as appropriate. See [Independent vs Contained vs Nested Subprograms], page 679, for some examples of these marker lines.

A compilation unit may also be a C-language source program, recognized as such by having a file extension of `.c` or an assembly-language program, recognized by its file extension of `.s`. In such a case, *COBOL* compilation of that file will be bypassed by the `cobc` command; instead, the file will be passed directly to the C compiler or assembler (executed automatically by `cobc`).

A compilation unit may *also* be an object-code module (output from the C compiler), recognized as such by having a file extension of `.o`. In these situations, all compilation will be bypassed, and the object code will be "bound" into the generated executable by the linker (an `ld` command executed internally by the `cobc` command).

Pre-compiled object-code subprograms may be automatically located by the GnuCOBOL compiler and the loader by using the `LD_LIBRARY_PATH` compilation-time environment variable (see [Compilation Time Environment Variables], page 645). If they are locatable through that environment variable, they need not be named on the `cobc` command.

The collection of compilation units supplied on a *single* `cobc` execution constitute a *Compilation Group*. All executable code produced from a single compilation group will be collected together into a single executable file, whose filename will be the same as that of the first compilation unit specified on the `cobc` command.

The simplest mode of compilation is to generate a single executable file from one or more GnuCOBOL source files:

```
cobc -x mainprog.cbl sub1.cbl sub2.cbl
```

The main program must be the first program found in the first compilation unit (`mainprog.cbl`). The remainder of that compilation unit as well as the rest of the files in the compilation group (`sub1.cbl` and `sub2.cbl`) must be independent and/or contained subprograms (see [Independent vs Contained vs Nested Subprograms], page 679).

This command assumes that all source files are in the directory from which the `cobc` command was executed. You are, of course, free to include full pathnames with any filename, if necessary.

With the `-x` switch on the compiler command, a single directly-executable executable file (UNIX, Windows/Cygwin, OSX) or "exe" file (Windows, Windows/MinGW) will be generated. This executable file has the compiled code for all COBOL programs contained within the compilation group specified on the `cobc` command included in the file.

Any subroutines or user-defined functions that weren't included in any of the source files comprising the compilation group will be treated as dynamically loadable subprograms (see [Dynamic vs Static Subprograms], page 681).

Optionally, the `-o` switch may be used in addition to `-x` to specify the name of the generated executable file. If `-o` switch is not specified, the filename of the 1st compilation unit will be used as the name of the executable file. The appropriate extension for the generated file (`.exe`, on a Windows computer, for example) will be added to the filename that is explicitly specified or implicitly assumed for the output file.

Compilations may be performed to generate dynamically-loadable modules (or dynamically-loadable libraries, as they are frequently called). These compilations are performed by using the `-m` switch instead of `-x` switch:

```
cobc -m mainprog.cbl sub1.cbl sub2.cbl
```

When the `-m` switch is used, an operating-system-specific dynamically-loadable module is generated *for each individual compilation unit*, using the filename of each compilation unit as its module filename and either an extension of `.so` (UNIX, Windows/Cygwin), `.dylib` (OSX) or `.dll` (Windows, Windows/MinGW).

You may compile GnuCOBOL subprograms into assembler source code which can then be assembled and linked with a main program when that main program is compiled. To create such an assembler source file, compile the subprogram(s) as follows:

```
cobc -S sprog1.cbl
```

The above generates an assembler source file named `sprog1.s`. If you have multiple subprograms to compile this way, just string their file names out on the command. Each will be translated to its own assembler source file.

Later, when you wish to compile a calling program and combine any needed assembly language subroutines in (as static subroutines — see [Dynamic vs Static Subprograms], page 681), use a command such as this:

```
cobc -x mainprog.cbl sprog1.s
```

10.1.1.1. cobc option -Xref an example

The following shows the output from using -Xref.

```
   Using the internal -Xref as extra options -X -t prog.list :

GnuCOBOL 3.1.2.0        prog.cbl             Thu Nov 25 15:31:52 2021  Page 0001

LINE    PG/LN  A...B.....................................................

000001  000001 IDENTIFICATION   DIVISION.
000002  000002 PROGRAM-ID.      prog.
000003  000003 ENVIRONMENT DIVISION.
000004  000004 CONFIGURATION SECTION.
000005  000005 DATA             DIVISION.
000006  000006 WORKING-STORAGE  SECTION.
000007  000007 COPY 'values.cpy'.
000001C 000001 78  I   VALUE 20.
000002C 000002 78  J   VALUE 5000.
000003C 000003 78  M   VALUE 5.
000008  000008 01  SETUP-REC.
000009  000009     05  FL1       PIC X(04).
000010  000010     05  FL2       PIC ZZZZZ.
000011  000011     05  FL3       PIC 9(04).
000012  000012     05  FL4       PIC 9(08) COMP.
000013  000013     05  FL5       PIC 9(04) COMP-4.
000014  000014     05  FL6       PIC Z,ZZZ.99.
000015  000015     05  FL7       PIC S9(05) SIGN LEADING SEPARATE.
000016  000016     05  FL8       PIC X(04).
000017  000017     05  FL9 REDEFINES FL8 PIC 9(04).
000018  000018     05  FLA.
000019  000019         10  FLB OCCURS I TIMES.
000020  000020             15  FLC PIC X(02).
000021  000021         10  FLD   PIC X(20).
000022  000022     05  FLD1      PIC X(100).
000023  000025     05  FLD3      PIC X(3).
000024  000026     05  FLD4      PIC X(4).
000025  000023     05  FLD2 OCCURS M TO J TIMES DEPENDING ON FL5.
000026  000024         10  FILLER PIC X(01).
000027  000027 PROCEDURE        DIVISION.
000028  000028     STOP RUN.

GnuCOBOL 3.1.2.0        prog.cbl             Thu Nov 25 15:31:52 2021  Page 0002

NAME                          DEFINED               REFERENCES

SETUP-REC                     8      referenced by child
FL1                           9      not referenced
FL2                           10     not referenced
```

```
FL3                             11     not referenced
FL4                             12     not referenced
FL5                             13       25                              x1
FL6                             14     not referenced
FL7                             15     not referenced
FL8                             16     not referenced
FL9                             17     not referenced
FLA                             18     not referenced
FLB                             19     not referenced
FLC                             20     not referenced
FLD                             21     not referenced
FLD1                            22     not referenced
FLD3                            23     not referenced
FLD4                            24     not referenced
FLD2                            25     not referenced


GnuCOBOL 3.1.2.0        prog.cbl             Thu Nov 25 15:31:52 2021  Page 0003

LABEL                           DEFINED                REFERENCES

E prog                          27

0 warnings in compilation group
0 errors in compilation group
```

10.1.1.2. Cross Reference listing using cobxref

This program is found in the contrib area or by itself in Sourceforge under its own name (`cobxref`).

The following shows the output from using `cobxref` with the same program example.

Note that cobxref also reports the COPY depth which can be up to 9 and includes the program being compiled as depth 1.

`Using cobxref with command cobxref prog.cbl :`

```
ACS Cobol Xref v2.02.03          Dictionary File for PROG    25/11/2021  15:22:00:04
                                  Page    1

     1  000001 IDENTIFICATION   DIVISION.
     2  000002 PROGRAM-ID.      prog.
     3  000003 ENVIRONMENT DIVISION.
     4  000004 CONFIGURATION SECTION.
     5  000005 DATA             DIVISION.
     6  000006 WORKING-STORAGE  SECTION.
     7  000007*COPY 'values.cpy'.
     8  000001 78  I   VALUE 20.
```

```
     9  000002 78  J   VALUE 5000.
    10  000003 78  M   VALUE 5.
    11  000008 01  SETUP-REC.
    12  000009     05  FL1       PIC X(04).
    13  000010     05  FL2       PIC ZZZZZ.
    14  000011     05  FL3       PIC 9(04).
    15  000012     05  FL4       PIC 9(08) COMP.
    16  000013     05  FL5       PIC 9(04) COMP-4.
    17  000014     05  FL6       PIC Z,ZZZ.99.
    18  000015     05  FL7       PIC S9(05) SIGN LEADING SEPARATE.
    19  000016     05  FL8       PIC X(04).
    20  000017     05  FL9 REDEFINES FL8 PIC 9(04).
    21  000018     05  FLA.
    22  000019         10  FLB OCCURS I TIMES.
    23  000020             15  FLC PIC X(02).
    24  000021         10  FLD   PIC X(20).
    25  000022     05  FLD1      PIC X(100).
    26  000025     05  FLD3      PIC X(3).
    27  000026     05  FLD4      PIC X(4).
    28  000023     05  FLD2 OCCURS M TO J TIMES DEPENDING ON FL5.
    29  000024         10  FILLER PIC X(01).
    30  000027 PROCEDURE        DIVISION.
    31  000028     STOP RUN.
    32        *>>>Info: Total Copy Depth Used = 02

ACS Cobol Xref v2.02.03          Dictionary File for PROG    25/11/2021  15:22:00:04

                                  Page    2
Symbols of Module: PROG (PROG)
------------------------------

Data Section (WORKING-STORAGE)  Defn     Locations
--------------------------------+-------------------------------------------------------

FL1                             000012W
FL2                             000013W
FL3                             000014W
FL4                             000015W
FL5                             000016W  000028
FL6                             000017W
FL7                             000018W
FL8                             000019W  000020
FL9                             000020W
FLA                             000021W
FLB                             000022W
FLC                             000023W
```

```
FLD                             000024W
FLD1                            000025W
FLD2                            000028W
FLD3                            000026W
FLD4                            000027W
I                               000008W  000022
J                               000009W  000028
M                               000010W  000028
SETUP-REC                       000011W

ACS Cobol Xref v2.02.03          Dictionary File for PROG    25/11/2021  15:22:00:04

                                  Page    3
Symbols of Module: PROG (PROG)
------------------------------
Procedure                       Defn     Locations
--------------------------------+-------------------------------------------------------------

None
```

10.1.2. Compilation Time Environment Variables

The following are the various environment variables that can play a role in the compilation of GnuCOBOL programs.

`COB_CC` * Set to the name of the C compiler you wish GnuCOBOL to use.

Use this feature at your own risk – you should always use the c compiler your gnucobol build was generated for.

`COB_CFLAGS` *
: Set to any switches that you'd like to pass on to the C compiler from the `cobc` compiler (in addition to any that `cobc` will specify).

`COB_CONFIG_DIR` *
: Set to the path to the folder where GnuCOBOL *config* files are kept.

`COB_COPY_DIR` *
: If copybooks your program needs are *not* stored in the same directory as your program, set this environment variable to the folder in which the copybooks may be found (IBM mainframe programmers will recognize this as `SYSLIB`).

`COB_LDADD`
: Set to any additional linker switches (`ld`) that can specify where standard libraries that must be linked with the program can be found. The default is `""` (empty).

`COB_LDFLAGS`
: Set to any linker (`ld`) switches that you'd like to pass on to the C compiler from the `cobc` compiler (in addition to any that cobc will specify).

`COB_LIBS` *
: Set to any linker switches (`ld`) that specify where standard libraries that must be linked with the program can be found.

`COBCPY`
: This environment variable provides an additional means of specifying where copybooks may be found by the compiler (see also COB_COPY_DIR, above).

`LD_LIBRARY_PATH`
: If you are planning on using static-linked subroutine libraries, set this variable to the path of the directory containing your libraries.

`TMPDIR`
`TMP`
: Set to a directory/folder appropriate to create temporary files in. The intermediate working files created by the compiler will be created here (and deleted once they're no longer needed).

  The variable `TMPDIR` is checked for a valid path first; if that isn't set, then `TMP` is checked.

  On a Windows system, the `TMP` environment variable is normally set for you when you logon. If you wish to use a different temporary folder, you may set `TMPDIR` yourself and have no fear of disrupting other Windows software that relies on TMP.

* The starred environment variables above have default values, established when the version of GnuCOBOL you are using was built. To see these default values, as well as other build-specific information, execute the command:

```
cobc -i
```

10.1.3. Predefined Compilation Variables

GnuCOBOL defines compilation variables when certain conditions are true.

If the condition associated with a variable is false, the variable is not defined during compilations.

`DEBUG`
: The `-d` debug flag is specified.

`EXECUTABLE`
: The module being compiled contains the main program.

`GCCOMP`
: The size of a `COMP` item is determined according to the GnuCOBOL scheme, where for a `PICTURE` of length:

1-2	item has 1 byte
3-4	item has 2 bytes
5-9	item has 4 bytes
10-18	item has 8 bytes.

`GNUCOBOL` GnuCOBOL is compiling the source unit.

`HOSTSIGNS`
: A signed packed-decimal item's value may be considered `NUMERIC` if the sign has value `X"F"`.

`IBMCOMP` The size of a `COMP` item is determined according to the IBM scheme, where for a `PICTURE` of length:

1-4	item has 2 bytes
5-9	item has 4 bytes
10-18	item has 8 bytes.

`MODULE` The module being compiled does not contain the main program.

`NOHOSTSIGNS`
: A signed packed-decimal item's value may not be considered

`NUMERIC` if the sign has value `X"F"`.

`NOIBMCOMP`
: The size of a `COMP` item is not determined according to the IBM scheme.

`NOSTICKYLINKAGE`
: Sticky-linkage (linkage-section items remaining allocated between invocations) is not enabled.

`NOTRUNC` Numeric data items are truncated according to their internal representation.

`P64` Pointers are greater than 32 bits long

`STICKY-LINKAGE`
: Sticky-linkage (linkage-section items remaining allocated between invocations) is enabled.

`TRUNC` Numeric data items are truncated according to their `PICTURE` clauses.

10.1.4. Locating Copybooks

The GnuCOBOL compiler will attempt to locate copybooks by searching for them in the following folders. The search will occur in the sequence shown below, and will terminate once a copybook is found.

1. The folder named as the *library-name-1* on the `COPY` statement (see [COPY], page 63).
2. The folder in which the program being compiled resides.
3. The folder named on the `-I` switch.
4. Each of the folders named on the `COBCPY` compilation-time environment variable (see [Compilation Time Environment Variables], page 645).

   A single folder may be named or multiple folders may be specified, separated by a system-appropriate delimiter character. When multiple folders are specified, they will be searched in the order they are named on the environment variable.

   If the GnuCOBOL compiler you are using was built to utilize a native Windows environment, use a semicolon (';') as the delimiter character.

   If, however, the GnuCOBOL compiler was built for a Unix, OSX or Linux environment, or was built for a Windows environment utilizing either the Cygwin or MinGW Unix emulators, use a colon character (':') as the delimiter.
5. The *single* folder specified on the COB_COPY_DIR environment variable.

As each of the above folders is searched for a copybook — `COPY` *`XXXXXXXX`*., for example — the GnuCOBOL compiler will attempt to locate the copybook file by any of the following names, in the sequence shown:

`XXXXXXXX``.CPY`
`XXXXXXXX``.CBL`
`XXXXXXXX``.COB`
`XXXXXXXX``.cpy`
`XXXXXXXX``.cbl`
`XXXXXXXX``.cob`
`XXXXXXXX`

The `COPY` statement is case-sensitive on UNIX systems; `COPY copybookname` and `COPY COPYBOOKNAME` will both fail to locate the `CopyBookName` copybook on a UNIX system.

Windows implementations of GnuCOBOL may, or may not, be similarly case sensitive with regard to copybook names, depending upon the Windows version and GnuCOBOL build options — it is safest to simply treat the COPY command as case-sensitive in all environments.

It is possible, however, to automatically cause all `COPY` statements to "fold" the names of all copybooks to upper-case by specifying the `-ffold-copy` switch with the `upper` option (i.e. `--fold-copy=upper`) to the GnuCOBOL compiler. Similarly, names could be folded to lower-case by using the `lower` option (i.e. `--fold-copy=lower`. If copybook libraries are maintained entirely using upper- or lower-case file names and extensions, either of these options will allow copybooks to be found regardless of how the programmer entered their names on `COPY` statements.

Case-folding may also be turned on and off within the program source code using the CDF `>>SET` statement (see [>>SET], page 71).

10.1.5. Compiler Configuration Files

GnuCOBOL uses compiler configuration files to define various options that will control the compilation process. These configuration files are specified using the `-conf` switch compilation switch and are found in the folder defined by the `COB_CONFIG_DIR` compilation-time environment variable (see [Compilation Time Environment Variables], page 645).

If this is not defined under *nix it will default to `/usr/local/share/gnucobol/config`.

The following is a verbatim listing of the default configuration file (the one used if you don't specify the `-conf` switch), just to show you the types of settings that may appear and taken from v3.1.2 :

```
# GnuCOBOL compiler configuration
#
# Copyright (C) 2001-2012, 2014-2020 Free Software Foundation, Inc.
# Written by Keisuke Nishida, Roger While, Simon Sobisch, Edward Hart,
# Ron Norman
#
# This file is part of GnuCOBOL.
#
# The GnuCOBOL compiler is free software: you can redistribute it
# and/or modify it under the terms of the GNU General Public License
# as published by the Free Software Foundation, either version 3 of the
# License, or (at your option) any later version.
#
# GnuCOBOL is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GnuCOBOL.  If not, see <https://www.gnu.org/licenses/>.


# Value: any string
name: "GnuCOBOL"

# Value: enum
standard-define                 0
#        CB_STD_GC = 0,
#        CB_STD_MF,
#        CB_STD_IBM,
#        CB_STD_MVS,
#        CB_STD_BS2000,
#        CB_STD_ACU,
```

```
#        CB_STD_85,
#        CB_STD_2002,
#        CB_STD_2014

# Value: int
tab-width:                      8
text-column:                    72
# Maximum word-length for COBOL words / Programmer defined words
# Be aware that GC checks the word length against COB_MAX_WORDLEN
# first (currently 63)
word-length:                    63

# Maximum literal size in general
literal-length:                 8191

# Maximum numeric literal size (absolute maximum: 38)
numeric-literal-length:         38

# Maximum number of characters allowed in the character-string (max. 255)
pic-length:                     255

# Default assign type
# Value: 'dynamic', 'external'
assign-clause:                  dynamic

# If yes, file names are resolved at run time using
# environment variables.
# For example, given ASSIGN TO "DATAFILE", the file name will be
#  1. the value of environment variable 'DD_DATAFILE' or
#  2. the value of environment variable 'dd_DATAFILE' or
#  3. the value of environment variable 'DATAFILE' or
#  4. the literal "DATAFILE"
# If no, the value of the assign clause is the file name.
#
filename-mapping:               yes

# Alternate formatting of numeric fields
pretty-display:                 yes

# Allow complex OCCURS DEPENDING ON
complex-odo:                    no

# Allow REDEFINES to other than last equal level number
indirect-redefines:             no

# Binary byte size - defines the allocated bytes according to PIC
```

```
# Value:         signed  unsigned  bytes
#                ------  --------  -----
# '2-4-8'         1 -  4    same        2
#                 5 -  9    same        4
#                10 - 18    same        8
#
# '1-2-4-8'       1 -  2    same        1
#                 3 -  4    same        2
#                 5 -  9    same        4
#                10 - 18    same        8
#
# '1--8'          1 -  2    1 -  2      1
#                 3 -  4    3 -  4      2
#                 5 -  6    5 -  7      3
#                 7 -  9    8 -  9      4
#                10 - 11   10 - 12      5
#                12 - 14   13 - 14      6
#                15 - 16   15 - 16      7
#                17 - 18   17 - 18      8
#
binary-size:                    1-2-4-8

# Numeric truncation according to ANSI
binary-truncate:                yes

# Binary byte order
# Value: 'native', 'big-endian'
binary-byteorder:               big-endian

# Allow larger REDEFINES items
larger-redefines-ok:            no

# Allow certain syntax variations (eg. REDEFINES position)
relax-syntax-checks:            no

# Allow zero length reference-modification
# (only checked with active EC-BOUND-REF-MOD)
ref-mod-zero-length:            yes

# Perform type OSVS - If yes, the exit point of any currently
# executing perform is recognized if reached.
perform-osvs:                   no

# Compute intermediate decimal results like IBM OSVS
arithmetic-osvs:                no
```

```
# MOVE like IBM (mvc); left to right, byte by byte
move-ibm:                       no

# SELECT RELATIVE KEY and ASSIGN fields must be in WORKING-STORAGE
select-working:         no

# LOCAL-STORAGE SECTION implies RECURSIVE attribute
local-implies-recursive:                no

# If yes, LINKAGE SECTION items remain allocated
# between invocations.
sticky-linkage:                 no

# If yes, allow non-matching level numbers
relax-level-hierarchy:          no

# If yes, evaluate constant expressions at compile time
constant-folding:               yes

# Allow Hex 'F' for NUMERIC test of signed PACKED DECIMAL field
hostsign:                       no

# If yes, set WITH UPDATE clause as default for ACCEPT dest-item,
# except if WITH NO UPDATE clause is used
accept-update:                  no

# If yes, set WITH AUTO clause as default for ACCEPT dest-item,
# except if WITH TAB clause is used
accept-auto:                    no

# If yes, DISPLAYs and ACCEPTs are, by default, done on the CRT (i.e., using
# curses).
console-is-crt:                 no

# If yes, allow redefinition of the current program's name. This prevents its
# use in a prototype-format CALL/CANCEL statement.
program-name-redefinition:      yes

# If yes, NO ECHO/NO-ECHO/OFF is the same as SECURE (hiding input with
# asterisks, not spaces).
no-echo-means-secure:           no

# If yes, the first item in a field screen ACCEPT/DISPLAY (e.g. DISPLAY x UPON
# CRT) is located after the previous ACCEPT/DISPLAY (as though LINE 0 COL 0 had
# been specified).
line-col-zero-default:          yes
```

```
# If yes, DISPLAY SPACES acts as ERASE EOS, DISPLAY X"01" acts as ERASE EOL,
# DISPLAY X"02" acts as BLANK SCREEEN and DISPLAY X"07" acts as BELL. Note
# DISPLAY LOW-VALUE is excluded from this; it will always just position the
# cursor.
display-special-fig-consts:     no

# If yes, COMP-1 is a signed 16-bit integer and any PICTURE clause is ignored.
binary-comp-1:                  no

# If yes, POINTER is handled as BINARY-DOUBLE UNSIGNED instead of its own class
numeric-pointer:                no

# auto-adjust to zero like MicroFocus does
move-non-numeric-lit-to-numeric-is-zero: no

# If yes, implicitly define a variable for an ASSIGN DYNAMIC which does not
# match an existing data item.
implicit-assign-dynamic-var:    yes

# What rules to apply to SCREEN SECTION items clauses
screen-section-rules:           gc

# Whether DECIMAL-POINT IS COMMA has effect in XML/JSON GENERATE
dpc-in-data:                    xml

# Dialect features
# Value: 'ok', 'warning', 'archaic', 'obsolete', 'skip', 'ignore', 'error',
#        'unconformable'

alter-statement:                        obsolete
comment-paragraphs:                     obsolete
call-overflow:                          archaic
data-records-clause:                    obsolete
debugging-mode:                         ok
use-for-debugging:                      ok
listing-statements:                     skip    # may be a user-defined word
title-statement:                        skip    # may be a user-defined word
entry-statement:                        ok
goto-statement-without-name:            obsolete
label-records-clause:                   obsolete
memory-size-clause:                     obsolete
move-noninteger-to-alphanumeric:        error
move-figurative-constant-to-numeric:    archaic
move-figurative-space-to-numeric:       error
move-figurative-quote-to-numeric:       obsolete
```

```
multiple-file-tape-clause:              obsolete
next-sentence-phrase:                   archaic
odo-without-to:                         warning
padding-character-clause:               obsolete
section-segments:                       ignore
stop-literal-statement:                 obsolete
stop-identifier-statement:              obsolete
same-as-clause:                         ok
type-to-clause:                         ok
usage-type:                                     ok
synchronized-clause:                    ok
special-names-clause:                   ok
top-level-occurs-clause:                ok
value-of-clause:                        obsolete
numeric-boolean:                        ok
hexadecimal-boolean:                    ok
national-literals:                      ok
hexadecimal-national-literals:          ok
national-character-literals:            warning
# TO-DO: Add separate config option for H"..." to be unsupported,numeric,non-numeric(acu)
acu-literals:                           unconformable
hp-octal-literals:                      unconformable
word-continuation:                      warning
not-exception-before-exception:         ok
accept-display-extensions:              ok
renames-uncommon-levels:                ok
symbolic-constant:                      ok
constant-78:                            ok
constant-01:                            ok
perform-varying-without-by:             ok
reference-out-of-declaratives:          warning
program-prototypes:                     ok
call-convention-mnemonic:               ok
call-convention-linkage:                ok
numeric-value-for-edited-item:          ok
incorrect-conf-sec-order:               ok
define-constant-directive:              archaic
free-redefines-position:                warning
records-mismatch-record-clause  warning
record-delimiter:                       ok
sequential-delimiters:                  ok
record-delim-with-fixed-recs:           ok
missing-statement:                      warning
zero-length-literals:                   ok
xml-generate-extra-phrases:             ok
continue-after:                         ok
```

```
goto-entry:                             warning
assign-variable:                        ok
assign-using-variable:                  ok
assign-ext-dyn:                         ok
assign-disk-from:                       ok
vsam-status:                            ignore

# use complete word list; synonyms and exceptions are specified below
reserved-words:         default

# not-reserved:
# Value: Word to be taken out of the reserved words list
not-reserved:   TERMINAL
# reserved:
#   Entries of the form word-1=word-2 define word-1 as an alias for default
# reserved word word-2. No spaces are allowed around the equal sign.
reserved:       AUTO-SKIP=AUTO
reserved:       AUTOTERMINATE=AUTO
reserved:       BACKGROUND-COLOUR=BACKGROUND-COLOR
reserved:       BEEP=BELL
reserved:       BINARY-INT=BINARY-LONG
reserved:       BINARY-LONG-LONG=BINARY-DOUBLE
reserved:       CELLS=CELL
reserved:       COLOURS=COLORS
reserved:       EMPTY-CHECK=REQUIRED
reserved:       EQUALS=EQUAL
reserved:       FOREGROUND-COLOUR=FOREGROUND-COLOR
reserved:       HIGH-VALUES=HIGH-VALUE
reserved:       INITIALISE=INITIALIZE
reserved:       INITIALISED=INITIALIZED
reserved:       LENGTH-CHECK=FULL
reserved:       LOW-VALUES=LOW-VALUE
reserved:       ORGANISATION=ORGANIZATION
reserved:       PIXELS=PIXEL
reserved:       SYNCHRONISED=SYNCHRONIZED
reserved:       TIMEOUT=TIME-OUT
reserved:       VALUES=VALUE
reserved:       ZEROES=ZERO
reserved:       ZEROS=ZERO
```

10.2. Running Programs

Once GnuCOBOL programs have been compiled into either directly-executable programs (created via the `-x` switch) or dynamically-loadable libraries (created via the `-m` switch), those programs may be executed from any shell environment. The exact manner in which the two are executed will differ, as described in the upcoming sections.

10.2.1. Direct Execution

GnuCOBOL programs compiled with the `-x` switch will be generated as directly-executable programs. For example, a native Windows or Windows/MinGW build of GnuCOBOL will generate an `.exe` file when the `-x` switch switch is specified to the compiler.

On Unix, OSX, or Windows/Cygwin builds, the `-x` switch switch will generate an executable binary file, usually with no particular extension unless one is explicitly requested of the compiler via the `-o` switch.

On a UNIX system this means the programs may be executed from a command shell such as bash, csh, ksh and so forth. When a GnuCOBOL program runs on a Windows system, it runs within a console window (i.e. `cmd.exe`). OSX versions of GnuCOBOL programs run within a `terminal.app` window.

Interactions between the program and the user will take place using the standard input, standard output and standard error streams. Any screen section I/O performed by the program will take place within the terminal window.

Direct program execution syntax is `[`*path*`]`*program* `[`*arguments*`]`.

For example:

```
/usr/local/printaccount ACCT=6625378
```

or

```
C:\\Users\\Me\\Documents\\Programs\\printaccount.exe ACCT=6625378
```

10.2.2. Executing Dynamically-Loadable Libraries

As discussed previously, dynamically-loadable libraries are created via the compiler's `-m` switch. Once so created, the program(s) in these libraries are executed from the command line (via the GnuCOBOL `cobcrun` utility), or as dynamically-loadable subprograms.

10.2.2.1. cobcrun - Command-line Execution

It is possible to generate executable modules for all GnuCOBOL programs, not just subprograms, by choosing to use the `-m` switch option to specify the loader output format, even for main programs.

Some may prefer to compile their GnuCOBOL main programs into these dynamically-loadable modules in the interests of using the same general compilation command for all programs without having to think "Is it a main program or a subprogram?".

Main programs compiled in this manner should be executed as follows:

`[`*path*`]cobcrun` *program* `[`*arguments*`]`

Do not specify the `.so` or `.dll` extension on the program name. The *program* value must exactly match the primary entry-point name of the main program (including upper-

and lower-case letters), unless you are planning on using *Call Folding* (see [Dynamically Loaded Subprograms], page 657).

The general usage and syntax of cobcrun is as follows as issued by running cobcrun -h (or –help) :

```
GnuCOBOL module loader

Usage: cobcrun [options] PROGRAM [parameter ...]
  or:  cobcrun options

Options:
  -h, -help                       display this help and exit
  -V, -version                    display cobcrun and runtime version and exit
  -i, -info                       display runtime information (build/environment)
  -v, -verbose                    display extended output with --info
  -c <file>, -config=<file>       set runtime configuration from <file>
  -r, -runtime-config             display current runtime configuration
                                  (value and origin for all settings)
  -M <module>, -module=<module>   set entry point module name and/or load path
                                  where -M module prepends any directory to the
                                  dynamic link loader library search path
                                  and any basename to the module preload list
                                  (COB_LIBRARY_PATH and/or COB_PRELOAD)
```

Here are two examples of using `cobcrun`. First, on a Unix, OSX or Windows/Cygwin system:

```
cd /usr/local
cobcrun printaccount acct=6625378
```

Or, on a Native Windows or Windows/MinGW system:

```
cd C:\Users\Me\Documents\Programs
cobcrun printaccount.exe acct=6625378
```

Note how the `cobcrun` command does not allow a path to be specified with the program name — the directory in which the programs dynamically loadable module exists must either be the current directory or must be defined in the current `PATH`.

10.2.2.2. Dynamically Loaded Subprograms

Dynamically-loaded subprograms are executed (from a COBOL syntax point of view) just like any other subprograms. What makes them unique, however, is that they are loaded into memory only when they are actually used the first time during the execution of a program.

When a dynamically-loadable module needs to be loaded (because it is not already in memory from a previous subprogram execution), the dynamically-loadable library will be sought by libcob in each directory named in the library specified by the `COB_LIBRARY_PATH` run-time environment variable will be searched and if not found then. Finally, if it

still cannot be located, execution will be terminated with an error message (*libcob: Cannot find module 'xxxxxxxx'*). For speed performance purposes all such should therefore be stored within the directories pointed to by the environment variable `COB_LIBRARY_PATH` run-time environment variable.

The process of locating dynamically-loadable modules is case sensitive on UNIX systems; `CALL "dynsub"` and `CALL "DYNSUB"` will both fail to locate the `DynSub.so` library on a UNIX system.

Windows implementations of GnuCOBOL may, or may not, be similarly case sensitive with regard to library names, depending upon the Windows version and GnuCOBOL build options — it is safest to simply treat library names as case sensitive in all environments.

It is possible, however, to automatically cause all library names to 'fold' to upper-case by specifying the `-ffold-call` switch with the "upper" option (i.e. `--fold-call=upper`) to the GnuCOBOL compiler. Similarly, library names could be folded to lower-case by using the "lower" option (i.e. `--fold-call=lower`. If libraries are maintained entirely using upper- or lower-case file names, either of these options will allow libraries to be found regardless of how the programmer entered their names on `CALL` statements.

See [Sub-Programming], page 679, for a complete discussion of sub-programming.

10.2.3. Run Time Environment Variables

The following is a list of the various environment variables that can play a role in the execution of GnuCOBOL programs.

`COB_DISPLAY_WARNINGS`
: If set to a value of '`Y`', any run-time warnings (such as noting the implicit closing of open files when a `GOBACK` statement (see [GOBACK], page 334) or `STOP` statement (see [STOP], page 399) with the `RUN` option is executed) will be displayed. Any other value for this environment variable (including not setting the variable at all) will suppress such messages.

`COB_LIBRARY_PATH`
: At runtime, GnuCOBOL will attempt to locate and load any application dynamically loadable libraries used this variable that points to the directories specified, if it wasn't found there, using the `PATH` environment variable.

`COB_LOAD_CASE`
: If set to either `UPPER` or `LOWER`, this environment variable will internally convert referenced entry point names to either upper- or lower-case before initiating searches for dynamically loadable modules. The `UPPER` and `LOWER` values of the environment variable are actually case insensitive.

`COB_PHYSICAL_CANCEL`
: If set to '`Y`', '`y`' or '`1`', a `CANCEL` statement (see [CANCEL], page 298) will physically unload a subprogram dynamically loadable module. If set to anything else, a `CANCEL` statement (see [CANCEL], page 298) logically unloads a module so that subsequent use will re-initialize the module as if it had actually been reloaded, but the overhead of actually reloading the module will be avoided.

`COB_PRE_LOAD`
: If set to any non-null value, this variable will cause all dynamically loadable libraries to be loaded when the program begins execution (rather than searching for and loading the module upon first use).

`COB_SET_DEBUG`
: If a `USE FOR DEBUGGING` (see [DECLARATIVES], page 254) section exists, the code within it will be disabled unless this environment variable is set to a value of '`Y`', '`y`' or '`1`'.

`COB_SET_TRACE`
: If the `-ftrace` switch (trace procedures) or `-ftraceall` switch (trace procedures and statements) was used when the program was compiled, setting this environment variable to a value of '`Y`' will activate the trace at the point the program begins execution. Setting this environment variable to any other value (or never setting it to ANY value) will disable tracing. Tracing, if configured by one of the two switches described above, can also be controlled via the the `READY TRACE` statement (see [READY TRACE], page 370) and `RESET TRACE` statement (see [RESET TRACE], page 372). If COB_SET_TRACE is set to Y, then tracing will always occur regardless of the presence of READY TRACE or RESET TRACE so in effect they will have no action on program execution.

`COB_SCREEN_ESC`
: If set to any non-blank value, this variable allows a `ACCEPT data-item` statement (see [ACCEPT data-item], page 272) to detect the `Esc` key.

`COB_SCREEN_EXCEPTIONS`
: Setting this variable to any non-blank value will allow the `ACCEPT data-item` statement (see [ACCEPT data-item], page 272) to detect the pressing of the `Esc`, `PgUp` and `PgDn` keys.

`COB_SORT_MEMORY`
: The value of this variable (an integer) will be used to define how much memory will be allocated for use in sorting. If the value is 1048576 or greater, that value will be used "as is" as the amount of memory (in bytes) to allocate. If the value is less than 1048576, the value will specify how many MB of memory will be allocated. The default sort memory amount is 128 MB.

`COB_SWITCH_n`
: (n=0 to 15); These environment variables correspond to `SWITCH-0` through `SWITCH-15`, defined in the `SPECIAL-NAMES` (see [SPECIAL-NAMES], page 92) paragraph. Setting them to `ON` will activate them; any other value turns them off.

`COB_SYNC` If set to a value of upper- or lower-case '`p`', this variable will force a file commit every time a file is written to (ensuring that data is immediately written to the file rather than retained in memory until a future commit occurs). This will slow-down update access to files, but will provide for better integrity in the event of a program failure.

`COB_TRACE_FILE`
: If set to any non-null value, this environment variable specifies the file to which all `-ftrace` switch and `-ftraceall` switch output will be written. If this is NOT set to a value, all `-ftrace` switch and `-ftraceall` switch output will be written to STDERR, where it may be piped via a "2> filename" on the command that executes the program.

`DB_HOME`
: If your GnuCOBOL build uses the Berkeley Database (BDB) package, use this environment variable to specify the folder in which the lock management files to be associated with all non-SORT files opened by the program will be stored. `ORGANIZATION INDEXED` (see [ORGANIZATION INDEXED], page 115) files will also have their data file allocated in the folder pointed to by this environment variable, if it exists.. Having this variable defined will activate record locking features on the `READ` statement (see [READ], page 366), `REWRITE` statement (see [REWRITE], page 374) and `WRITE` statement (see [WRITE], page 417). Even with DB_HOME, locking will not work with `ORGANIZATION SEQUENTIAL` (see [ORGANIZATION SEQUENTIAL], page 109), `ORGANIZATION LINE SEQUENTIAL` (see [ORGANIZATION LINE SEQUENTIAL], page 111) or ORGANIZATION RELATIVE files with GnuCOBOL builds created for Windows/MinGW. `ORGANIZATION INDEXED` locks will work with Windows/MinGW + BDB and all locks will work for all file organizations with UNIX GnuCOBOL builds.

`PATH`
: The GnuCOBOL "bin" directory should be defined in the PATH.

`TMPDIR`
`TMP`
`TEMP`
: One of these environment variables must be set to a directory/folder appropriate to create temporary files in. They will be checked in the order shown. This will be used by the `SORT` statement (see [SORT], page 391) and `MERGE` statement (see [MERGE], page 349) to create temporary work files. You may also use this folder for any temporary files your application may require.

Also used during execution of programs is runtime.cfg also found in `/usr/local/share/gnucobol/config` for *nix and this file can also be changed to match your environment if needed. When viewing, note the Default settings.

10.2.3.1. General instructions

```
The initial runtime.cfg file is found in the  $COB_CONFIG_DIR/config
(  COB_CONFIG_DIR  defaults to  installdir/gnucobol  ).
The environment variable  COB_RUNTIME_CONFIG  may define a different runtime
configuration file to read.

If settings are included in the runtime environment file multiple times
then the last setting value is used, no warning occurs.

Settings via environment variables always take precedence over settings
that are given in runtime configuration files. And the environment is
```

```
checked after completing processing of the runtime configuration file(s)

All values set to string variables or environment variables are checked
for  ${envvar}  and replacement is done at the time of the setting.

Any environment variable may be set with the directive  setenv  .
Example:  setenv COB_LIBRARY_PATH ${LD_LIBRARY_PATH}

Any environment variable may be unset with the directive  unsetenv
(one var per line).
Example:  unsetenv COB_LIBRARY_PATH

Runtime configuration files can include other files with the directive
include.
Example:  include my-runtime-configuration-file

To include another configuration file only if it is present use the directive
includeif.
You can also use  ${envvar}  inside this.
Example:  includeif ${HOME}/mygc.cfg

If you want to reset a parameter to its default value use:
   reset parametername

Most runtime variables have boolean values, some are switches, some have
string values, integer values and some are size values.
The boolean values will be evaluated as following:
      to true:          1, Y, ON, YES, TRUE  (no matter of case)
      to false:         0, N, OFF
A 'size' value is an integer optionally followed by K, M, or G for kilo, mega
or giga.

For convenience a parameter in the runtime.cfg file may be defined by using
either the environment variable name or the parameter name.
In most cases the environment variable name is the parameter name (in upper
case) with the prefix  COB_  .

Note: If you want to *slightly* speed up a program's startup time, remove all
      of the comments from the actual real configuration file that is processed
```

10.2.3.2. General Environment

```
Environment name:  COB_DISABLE_WARNINGS
  Parameter name:  disable_warnings
         Purpose:  turn off runtime warning messages
```

```
            Type:  boolean
         Default:  false
         Example:  DISABLE_WARNINGS  TRUE

Environment name:  COB_ENV_MANGLE
  Parameter name:  env_mangle
         Purpose:  names checked in the environment would get non alphanumeric
                   change to '_'
            Type:  boolean
         Default:  false
         Example:  ENV_MANGLE  TRUE

Environment name:  COB_SET_DEBUG
  Parameter name:  debugging_mode
         Purpose:  to enable USE ON DEBUGGING procedures that were active
                   during compile-time because of WITH DEBUGGING MODE,
                   otherwise the code generated will be skipped
            Type:  boolean
         Default:  false
         Example:  COB_SET_DEBUG  1

Environment name:  COB_SET_TRACE
  Parameter name:  set_trace
         Purpose:  to enable COBOL trace feature
            Type:  boolean
         Default:  false
         Example:  SET_TRACE  TRUE

Environment name:  COB_TRACE_FILE
  Parameter name:  trace_file
         Purpose:  to define where COBOL trace output should go
            Type:  string
         Default:  stderr
         Example:  TRACE_FILE  ${HOME}/mytrace.log

Environment name:  COB_TRACE_FORMAT
  Parameter name:  trace_format
         Purpose:  to define format of COBOL trace output
            Type:  string
         Default:  "%P %S Line: %L"
                   %P is replaced by Program-Id/Function-Id minimal length 29
                      with prefix
                   %I is replaced by Program-Id/Function-Id variable length,
                      without prefix
                   %L is replaced by Line number, right justified, length 6
                   %S is replaced by statement type and name
```

```
                  %F is replaced by source file name
           Example:  TRACE_FORMAT  "Line: %L %S"
              Note:  format of GC2.2 and older:
                     "PROGRAM-ID: %I    Line: %L        %S"

For v4.0+
 Environment name:  COB_TRACE_IO
  Parameter name:  trace_io
         Purpose:  define if I/O details should be added to trace
            Type:  boolean
         Default:  false
         Example:  TRACE_IO true

 Environment name:  COB_DUMP_FILE
   Parameter name:  dump_file
          Purpose:  to define where COBOL dump output should go
             Note:  The -fdump=all compile option prepares for dump
             Type:  string       : $$ is replaced by process id
          Default:  stderr
          Example:  DUMP_FILE  ${HOME}/mytrace.log

 Environment name:  COB_DUMP_WIDTH
   Parameter name:  dump_width
          Purpose:  to define COBOL dump line length
             Type:  integer
          Default:  100
          Example:  dump_width 120

For v4.0+
 Environment name:  COB_STATS_RECORD
  Parameter name:  stats_record
         Purpose:  define if I/O statistics should be written
            Type:  boolean
         Default:  false
         Example:  STATS_RECORD true

For v4.0+
 Environment name:  COB_STATS_FILE
  Parameter name:  stats_file
         Purpose:  to define where COBOL I/O statistics should be written
                   The file is appended to
            Type:  string
         Default:  stderr
         Example:  STATS_FILE  ${HOME}/mystats.txt

 Environment name:  COB_CURRENT_DATE
```

```
  Parameter name:  current_date
         Purpose:  specify an alternate Date/Time to be returned to ACCEPT
                   clauses this is used for testing purposes or to tweak
                   a missing offset partial setting is allowed
            Type:  numeric string in format YYYYDDMMHH24MISS or date string
         Default:  the operating system date is used
         Example:  COB_CURRENT_DATE "2016/03/16 16:40:52"
                   current_date YYYYMMDDHHMMSS+01:00
```

10.2.3.3. Call Environment

```
Environment name:  COB_LIBRARY_PATH
  Parameter name:  library_path
         Purpose:  paths for dynamically-loadable modules
            Type:  string
            Note:  the default paths .:/installpath/extras are always
                   added to the given paths
         Example:  LIBRARY_PATH    /opt/myapp/test:/opt/myapp/production

Environment name:  COB_PRE_LOAD
  Parameter name:  pre_load
         Purpose:  modules that are loaded during startup, can be used
                   to CALL COBOL programs or C functions that are part
                   of a module library
            Type:  string
            Note:  the modules listed should NOT include extensions, the
                   runtime will use the right ones on the various platforms,
                   COB_LIBRARY_PATH is used to locate the modules
         Example:  PRE_LOAD      COBOL_function_library:external_c_library

Environment name:  COB_LOAD_CASE
  Parameter name:  load_case
         Purpose:  resolve ALL called program names to UPPER or LOWER case
            Type:  Only use  UPPER  or  LOWER
         Default:  if not set program names in CALL are case sensitive
         Example:  LOAD_CASE  UPPER

Environment name:  COB_PHYSICAL_CANCEL
  Parameter name:  physical_cancel
         Purpose:  physically unload a dynamically-loadable module on CANCEL,
                   this frees some RAM and allows the change of modules during
                   run-time but needs more time to resolve CALLs (both to
                   active and not-active programs)
           Alias:  default_cancel_mode, LOGICAL_CANCELS (0 = yes)
            Type:  boolean (evaluated for true only)
```

```
          Default:  false
          Example:  PHYSICAL_CANCEL  TRUE
```

10.2.3.4. File I/O

```
 Environment name:  COB_VARSEQ_FORMAT
   Parameter name:  varseq_format
          Purpose:  declare format used for variable length sequential files
                    - different types and lengths precede each record
                    - 'length' is the data length, does not include the prefix
             Type:  0   means 2 byte record length (big-endian) + 2 NULs
                    1   means 4 byte record length (big-endian)
                    2   means 4 byte record length (local machine int)
                    3   means 2 byte record length (big-endian)
          Default:  0
          Example:  VARSEQ_FORMAT 1

For v4.0+
 Environment name: COB_VARREL_FORMAT
  Parameter name: varrel_format
         Purpose: declare format to be used for variable length relative files
            Type: gc  means 'size_t' record length (local machine) precedes
                            maxiumum length data record
                  mf  means file is in Micro Focus format
                 b32  means Big-Endian 32-bit 'int' record length precedes data
                 b64  means Big-Endian 64-bit 'int' record length precedes data
                 l32  means Little-Endian 32-bit 'int' record length precedes data
                 l64  means Little-Endian 64-bit 'int' record length precedes data
         Default: gc
            NOTE: 'gc' results in files which cannot be used if copied between
                       machines of different hardware archeticture
         Example: VARREL_FORMAT mf

For v4.0+
 Environment name:  COB_FIXREL_FORMAT
  Parameter name:  fixrel_format
         Purpose:  declare format to be used for fixed length relative
                   files (different types and lengths preceding each record)
            Type:  b32 means 4 byte record length (big-endian)
                   l32 means 4 byte record length (little-endian)
                   b64 means 8 byte record length (big-endian)
                   l64 means 8 byte record length (little-endian)
                   mf  means Micro Focus default
                   gc  means GnuCOBOL default (local 'size_t')
```

```
         Default:  gc  fixed size with no record length prefix
         Example:  FIXREL_FORMAT B32

For v4.0+
 Environment name: COB_VARFIX_FORMAT
  Parameter name: varfix_format
         Purpose: declare format to be used for fixed length relative files
            Type: gc  means 'size_t' record length (local machine) precedes
                            fixed length data record
                  mf  means file is in Micro Focus format
                 b32  means Big-Endian 32-bit 'int' record length precedes data
                 b64  means Big-Endian 64-bit 'int' record length precedes data
                 l32  means Little-Endian 32-bit 'int' record length precedes data
                 l64  means Little-Endian 64-bit 'int' record length precedes data
         Default: gc
            NOTE: 'gc' results in files which cannot be used if copied between
                       machines of different hardware archeticture
         Example: VARFIX_FORMAT mf

 Environment name:  COB_FILE_PATH
   Parameter name:  file_path
          Purpose:  define default location where data files are stored
             Type:  file path directory
          Default:  .  (current directory)
          Example:  FILE_PATH ${HOME}/mydata

 Environment name:  COB_LS_FIXED
   Parameter name:  ls_fixed
          Purpose:  Defines if LINE SEQUENTIAL files should be fixed length
                    (or variable, by removing trailing spaces)
            Alias:  STRIP_TRAILING_SPACES  (0 = yes)
             Type:  boolean
          Default:  false
          Example:  LS_FIXED TRUE

 Environment name:  COB_LS_NULLS
   Parameter name:  ls_nulls
          Purpose:  Defines for LINE SEQUENTIAL files what to do with data
                    which is not DISPLAY type.  This could happen if a LINE
                    SEQUENTIAL record has BINARY/COMP data fields in it.
                    For v4.0+ (may change before its release candidate)
                    This option is only for GnuCOBOL format files
             Type:  boolean
          Default:  false
             Note:  The TRUE setting will insert a null character x"00" before
                    those values to escape them, and redo on read-in.
```

```
          Example:  LS_NULL = TRUE

For v4.0+
Environment name:  COB_LS_SPLIT
  Parameter name:  ls_split
         Purpose:  Defines for LINE SEQUENTIAL files what to do when a record
                   is longer than the program handles. If 'ls_split=true' then
                   the data is returned as multiple records
            Type:  boolean
         Default:  false
                   The record is truncated and the file skips to the next LF
         Example:  LS_SPLIT = TRUE

For v4.0+
Environment name:  COB_LS_VALIDATE
  Parameter name:  ls_validate
         Purpose:  Defines for LINE SEQUENTIAL files that the data should be
                   validated. If any record has non-DISPLAY characters then
                   an error status of 34 is returned
                   For v4.0+ (may change before its release candidate)
                   This option is only for GnuCOBOL format files
            Type:  boolean
         Default:  true
            Note:  The TRUE setting does data validation
                   The FALSE setting lets non-DISPLAY characters be written
                   If LS_NULLS is set, then LS_VALIDATE is not checked
         Example:  LS_VALIDATE = FALSE

For v4.0+ (may be replaced before its release candidate)
 Environment name: COB_MF_FILES
  Parameter name: mf_files
         Purpose:  Declares that all files in the program should be in
                   Micro Focus compatible format.
            Type:  boolean (evaluated for true only)
         Default:  false
         Example:  mf_files True

For v4.0+ (may be replaced before its release candidate)
 Environment name:  COB_MF_LS_NULLS
  Parameter name:  mf_ls_nulls
         Purpose:  Defines for Micro Focus compatible LINE SEQUENTIAL files
                   what to do with data which is not DISPLAY type.
                   This could happen if a LINE SEQUENTIAL record has
                   BINARY/COMP data fields in it.
            Type:  boolean
         Default:  true
```

```
            Note:  The TRUE setting will handle files that contain COMP data
                   in a similar manner to the method used by Micro Focus COBOL
         Example:  MF_LS_NULLS = TRUE

For v4.0+ (may be replaced before its release candidate)
 Environment name:  COB_MF_LS_INSTAB
   Parameter name:  mf_ls_instab
          Purpose:  Defines for LINE SEQUENTIAL files that multiple spaces
                    should be replaced by a TAB character, assuming a 'tab set'
                    value of 8. Each TAB means to skip to the next column that
                    is a multiple of 8
                    Similar to Micro Focus INSERTTAB=ON option
             Type:  boolean
          Default:  false
          Example:  MF_LS_INSTAB = TRUE

For v4.0+ (may be replaced before its release candidate)
 Environment name:  COB_MF_LS_SPLIT
  Parameter name:  mf_ls_split
         Purpose:  Defines for Micro Focus compatible LINE SEQUENTIAL files what
                   to do when a record is longer than the program handles.
                   If 'mf_ls_split=true' then
                   the data is returned as multiple records
            Type:  boolean
         Default:  true
         Example:  MF_LS_SPLIT = FALSE

For v4.0+ (may be replaced before its release candidate)
 Environment name:  COB_MF_LS_VALIDATE
  Parameter name:  mf_ls_validate
         Purpose:  Defines for Micro Focus compatible LINE SEQUENTIAL files
                   that the data should be validated.
                   If any record has non-DISPLAY characters then
                   an error status of 34 is returned
            Type:  boolean
         Default:  false
            Note:  The TRUE setting does data validation
                   The FALSE setting lets non-DISPLAY characters be written
                   If MF_LS_NULLS is set, then MF_LS_VALIDATE is not checked
         Example:  MF_LS_VALIDATE = FALSE

For v4.0+
 Environment name:  COB_SHARE_MODE
  Parameter name:  share_mode
         Purpose:  Defines what file sharing option should be used
            Type:  -- choice of values ---
```

```
                  none - nothing overrides application code
                  read - files opened as SHARE READ ONLY
                  all  - files opened as SHARE ALL OTHERS
                  no   - files opened as SHARE NO OTHERS
         Default:  none
         Example:  share_mode = ALL

For v4.0+
 Environment name:  COB_RETRY_MODE
  Parameter name:  retry_mode
         Purpose:  Defines what I/O retry sharing option should be used
            Type:  --- choice of values ---
                   none    - nothing overrides application code
                   never   - I/O is never retried
                   forever - I/O will be retried until success
         Default:  none
         Example:  retry_mode = never

For v4.0+
 Environment name:  COB_RETRY_TIMES
  Parameter name:  retry_times
         Purpose:  Defines how many times I/O should be retried
            Type:  integer
         Default:  0
         Example:  retry_times = 10

For v4.0+
 Environment name:  COB_RETRY_SECONDS
  Parameter name:  retry_seconds
         Purpose:  Defines how many seconds I/O should be retried
            Type:  integer
         Default:  0
         Example:  retry_seconds = 6

For v4.0+
 Environment name:  COB_KEYCHECK
  Parameter name:  keycheck
         Purpose:  Must INDEXED file keys match COBOL SELECT exactly
            Type:  boolean
         Default:  true
         Example:  keycheck = off

 Environment name:  COB_SYNC
   Parameter name:  sync
          Purpose:  Should the file be synced to disk after each write/update
             Type:  boolean
```

```
          Default:  false
          Example:  SYNC: TRUE

 Environment name:  COB_SORT_MEMORY
   Parameter name:  sort_memory
          Purpose:  Defines how much RAM to assign for sorting data
                    if this size is exceeded the  SORT  will be done
                    on disk instead of memory
             Type:  size  but must be more than 1M
          Default:  128M
          Example:  SORT_MEMORY 64M

 Environment name:  COB_SORT_CHUNK
   Parameter name:  sort_chunk
          Purpose:  Defines how much RAM to assign for sorting data in chunks
             Type:  size  but must be within 128K and 16M
          Default:  256K
          Example:  SORT_CHUNK 1M
```

10.2.3.5. Screen I/O

```
 Environment name:  COB_BELL
   Parameter name:  bell
          Purpose:  Defines how a request for the screen to beep is handled
             Type:  FLASH, SPEAKER, FALSE, BEEP
          Default:  BEEP
          Example:  BELL SPEAKER

 Environment name:  COB_REDIRECT_DISPLAY
   Parameter name:  redirect_display
          Purpose:  Defines if DISPLAY output should be sent to 'stderr'
             Type:  boolean
          Default:  false
          Example:  redirect_display Yes

 Environment name:  COB_SCREEN_ESC
   Parameter name:  screen_esc
          Purpose:  Enable handling of ESC key during ACCEPT
             Type:  boolean
          Default:  false
             Note:  is only evaluated if COB_SCREEN_EXCEPTIONS is active
          Example:  screen_esc Yes

 Environment name:  COB_SCREEN_EXCEPTIONS
```

```
  Parameter name:  screen_exceptions
         Purpose:  enable exceptions for function keys during ACCEPT
            Type:  boolean
         Default:  false
         Example:  screen_exceptions Yes

Environment name:  COB_TIMEOUT_SCALE
  Parameter name:  timeout_scale
         Purpose:  specify translation in milliseconds for ACCEPT clauses
                   BEFORE TIME value / AFTER TIMEOUT
            Type:  integer
                   0 means 1000 (Micro Focus COBOL compatible), 1 means 100
                   (ACUCOBOL compatible), 2 means 10, 3 means 1
         Default:  0
         Example:  timeout_scale 3

Environment name:  COB_INSERT_MODE
  Parameter name:  insert_mode
         Purpose:  specify default insert mode for ACCEPT; 0=off, 1=on
            Type:  boolean
         Default:  false
            Note:  also sets the cursor type (if available)
         Example:  insert_mode Y

Environment name:  COB_MOUSE_FLAGS
 Parameter name:  mouse_flags
        Purpose:  specify which mouse events will be sent as function key
                  to the application during ACCEPT and how they will be
                  handled
           Type:  int (by bits)
        Default:  1
           Note:  0 disables the mouse cursor, any other value enables it,
                  any value containing 1 will enable internal handling (click
                  to position, double-click to enter).
                  See copy/screenio.cpy for list of events and their values.
          Alias:  MOUSE_FLAGS
        Example:  11 (enable internal handling => 1, left press => 2,
                      double-click => 8; 1+2+8=11)
Environment name:  COB_MOUSE_INTERVAL
 Parameter name:  mouse_interval
        Purpose:  specifies the maximum time (in thousands of a second)
                  that can elapse between press and release events for them
                  to be recognized as a click.
           Type:  int (0 - 166)
        Default:  100
           Note:  0 disables the click resolution (instead press + release
```

```
                  are recognized), also disables positioning by mouse click

 Environment name:  COB_DISPLAY_PRINT_PIPE
  Parameter name:  display_print_pipe
         Purpose:  Defines command line used for sending output of
                   DISPLAY UPON PRINTER to (via pipe)
                   This is very similar to Micro Focus COBPRINTER
            Note:  Each executed DISPLAY UPON PRINTER statement causes a
                   new invocation of command-line (= new process start).
                   Each invocation receives the data referenced in
                   the DISPLAY statement and is followed by an
                   end-of-file condition.
                   COB_DISPLAY_PRINT_FILE, if set, takes precedence
                   over COB_DISPLAY_PRINT_PIPE.
           Alias:  COBPRINTER
            Type:  string
         Default:  not set
         Example:  print 'cat >>/tmp/myprt.log'

 Environment name:  COB_DISPLAY_PRINT_FILE
  Parameter name:  display_print_file
         Purpose:  Defines file to be appended to by DISPLAY UPON PRINTER
            Note:  Each DISPLAY UPON PRINTER opens, appends and closes the file.
            Type:  string       : $$ is replaced by process id
         Default:  not set
         Example:  display_printer '/tmp/myprt.log'

 Environment name:  COB_DISPLAY_PUNCH_FILE
  Parameter name:  display_punch_file
         Purpose:  Defines file to be created on first
                   DISPLAY UPON SYSPUNCH/SYSPCH
            Note:  The file will be only be closed on runtime exit.
            Type:  string       : $$ is replaced by process id
         Default:  not set
         Example:  display_punch './punch_$$.out'

 Environment name:  COB_LEGACY
   Parameter name:  legacy
          Purpose:  keep behaviour of former runtime versions, currently only
                    for setting screen attributes for non input fields
             Type:  boolean
          Default:  not set
          Example:  legacy true

 Environment name:  COB_EXIT_WAIT
   Parameter name:  exit_wait
```

```
          Purpose:  to wait on main program exit if an extended screenio
                    DISPLAY was issued without an ACCEPT following
             Type:  boolean
          Default:  true
          Example:  COB_EXIT_WAIT off

 Environment name:  COB_EXIT_MSG
   Parameter name:  exit_msg
          Purpose:  string to display if COB_EXIT_WAIT is processed, set to '@w{}'
                    if no actual display but an ACCEPT should be done
             Type:  string
          Default:  'end of program, please press a key to exit' (localized)
          Example:  COB_EXIT_MSG '@w{}'
```

10.2.3.6. Report I/O

```
 Environment name:  COB_COL_JUST_LRC
   Parameter name:  col_just_lrc
          Purpose:  If true, then COLUMN defined as LEFT, RIGHT or CENTER
                    will have the data justified within the field limits
                    If false, then the data is just copied into the column as is
             Type:  boolean
          Default:  TRUE
          Example:  col_just_lrc True
```

10.2.3.7. File I/O Environment Variables and/or dictionary file

```
GnuCOBOL 4.+ only!

Before a file is opened a check is done for environment variables that
may define various attributes of the file
First a check is made for attributes for files of the same ORGANIZATION
IX_OPTIONS for INDEXED, SQ_OPTIONS for SEQUENTIAL, RL_OPTIONS for RELATIVE
LS_OPTIONS for LINE SEQUENTIAL, LA_OPTIONS for LINE ADVANCING SEQUENTIAL
If none of these are present, it then checks for IO_OPTIONS

Then an additional check is done for IO_asgnmame where 'asgname' was
the ASSIGN EXTERNAL name used in the program

The environment variable (or dictionary file) may contain any of the
following keywords, separated by spaces and/or commas

You can specify just the keyword and it is assumed to mean set to true,
```

```
or no-keyword (or no_keyword or nokeyword) which means set to false,
or keyword=true or keyword=false.  The valid keywords are:
Keyword          Meaning
=========        ======================================================
type=xx          Set file organization where 'xx' is one of
                 IX = INDEXED, SQ = SEQUENTIAL, RL = RELATIVE
                 LS = LINE SEQUENTIAL, LA = LINE ADVANCING
mf               Set file to Micro Focus compatible format
gc               Set to original GnuCOBOL default format
recsz            The size for fixed size record file
maxsz            Maximum record size for variable length records
minsz            Minimum record size for variable length records
ls_nulls         Do NUL insertion before characters less than a SPACE,
                 Default: false
ls_validate      Validate data for LINE Sequential Files, Default: true
crlf             Lines end with CR LF (Windows format for text files)
lf               Lines end with LF (Unix format for text files)
sync             Sync all writes to disk
B32              Use 32-bit Big-Endian format 'int' as record length
L32              Use 32-bit Little-Endian format 'int' as record length
B64              Use 64-bit Big-Endian format 'int' or 'size_t' as record length
L64              Use 64-bit Little-Endian format 'int' or 'size_t' as record length
trace            Enable I/O trace when program execution tracing is enabled
stats            Write I/O statistic information on file close
retry_times      Default number of times to retry I/O
retry_seconds    Number of seconds between I/O retry attempts
retry_forever    Retry I/O forever
retry_never      Never retry I/O operations
ignore_lock      Ignore record locks
advancing_lock   Advance to the next record if lock condition
share_all        Share file with ALL others
share_read       Share file for READ only
share_no         Share file with NO others

 ---- For INDEXED files -----
format=ixhandler INDEXED file format: CISAM,DISAM,VBISAM,BDB,LMDB
format=auto      INDEXED file format is determined by inspecting the file
nkeys=n          number of indexes
key1=(loc:len)   loc (zero relative) of key, len of key
key2=(loc:len,loc:len ...)
                 define composite index
dupn=Y           index allows dups

  ---- For INDEXED BDB files -----
big_endian       Set internal 'int' byte order to BIG ENDIAN
little_endian    Set internal 'int' byte order to LITTLE ENDIAN
```

10.2.4. Program Arguments

Regardless of the manner in which a main program is executed (i.e. directly or via `cobcrun`), any arguments specified to the program may be retrieved via any of the following:

- `ACCEPT FROM COMMAND-LINE` (see [ACCEPT FROM COMMAND-LINE], page 269)
- `PROCEDURE DIVISION CHAINING` (see [PROCEDURE DIVISION CHAINING], page 250)

10.3. Binary Truncation

By default, the GnuCOBOL compiler will truncate binary data items to the precision indicated by their `PICTURE` (see [PICTURE], page 194) clause, if they have one. This applies to COMP, BINARY and COMP-4 items Only.
The fact is, however, that binary truncation has a significant effect on the performance of GnuCOBOL programs. When binary truncation is in effect, arithmetic operations performed against all types of numeric data items (even `USAGE DISPLAY`) are slowed down.

Before continuing, it's worth making the point that we're *not* talking about astronomical performance degradations here. Today's computers are *fast*, and a user sitting at the keyboard, running a GnuCOBOL program is unlikely to notice. But, if you have a GnuCOBOL program that has to process large amounts of data, performing some significant "number crunching" against that data as it goes, the impact of truncation could become noticeable.

The following program compares the performance of performing arithmetic operations (in a totally non-scientific, non-rigorous way) against data items with a `USAGE` (see [USAGE], page 232) of `DISPLAY`, `COMP` and `BINARY-LONG`. It was actually my intent when I first wrote the program to merely demonstrate the relative performance differences between different types of numeric data storage, and it certainly met that objective.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DEMOMATH.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Begin-Time.
    05 BT-HH                    PIC 9(2).
    05 BT-MM                    PIC 9(2).
    05 BT-SS                    PIC 9(2).
    05 BT-HU                    PIC 9(2).
01  Binary-Item         BINARY-LONG SIGNED VALUE 0.
01  Comp-Item           COMP    PIC S9(9)  VALUE 0.
01  Display-Item        DISPLAY PIC S9(9)  VALUE 0.
01  End-Time.
    05 ET-HH                    PIC 9(2).
    05 ET-MM                    PIC 9(2).
    05 ET-SS                    PIC 9(2).
    05 ET-HU                    PIC 9(2).
```

```
78  Repeat-Count                VALUE 10000000.
01  Time-Diff                   PIC ZZ9.99.
PROCEDURE DIVISION.
010-Test-Usage-DISPLAY.
    ACCEPT  Begin-Time FROM TIME
    PERFORM Repeat-Count TIMES
        ADD 7 TO Display-Item
    END-PERFORM
    PERFORM 100-Determine-Time-Diff
    DISPLAY 'USAGE DISPLAY: ' Time-Diff ' SECONDS'
    .
020-Test-Usage-COMP.
    ACCEPT Begin-Time FROM TIME
    PERFORM Repeat-Count TIMES
        ADD 7 TO Comp-Item
    END-PERFORM
    PERFORM 100-Determine-Time-Diff
    DISPLAY 'USAGE COMP:    ' Time-Diff ' SECONDS'
    .
040-Test-Usage-BINARY.
    ACCEPT Begin-Time FROM TIME
    PERFORM Repeat-Count TIMES
        ADD 7 TO Binary-Item
    END-PERFORM
    PERFORM 100-Determine-Time-Diff
    DISPLAY 'USAGE BINARY:  ' Time-Diff ' SECONDS'
    .
099-Done.
    STOP RUN
    .
100-Determine-Time-Diff.
    ACCEPT End-Time FROM TIME
    COMPUTE Time-Diff =
      ( (ET-HH * 360000 + ET-MM * 6000 + ET-SS * 100 + ET-HU)
      - (BT-HH * 360000 + BT-MM * 6000 + BT-SS * 100 + BT-HU) )
      / 100
    .
```

Each data item has 7 added to it ten *million* times.

The time (to one-one-hundredth of a second) will be retrieved before and after each test and the difference between the two is displayed. This is why the computations were done so many times — it was to make sure the timing was measurable with only a 1/100 second "stopwatch".

I also ran the tests multiple times, just to make sure I had consistent results (I did). Like I mentioned earlier, this is not a rigorous, scientific benchmark of numeric performance; it's just a quick-and-dirty comparison.

Here are the results:

```
Test 1:
USAGE DISPLAY:   1.72 SECONDS
USAGE COMP:      0.62 SECONDS
USAGE BINARY:    0.02 SECONDS

Test 2:
USAGE DISPLAY:   1.69 SECONDS
USAGE COMP:      0.61 SECONDS
USAGE BINARY:    0.02 SECONDS

Test 3:
USAGE DISPLAY:   1.69 SECONDS
USAGE COMP:      0.65 SECONDS
USAGE BINARY:    0.02 SECONDS
```

The results I saw here were consistent with those that would have been obtained from most of the COBOL implementations I have ever worked with — `USAGE COMP` has a significant performance advantage over `USAGE DISPLAY` and `USAGE BINARY-LONG` (and presumably the other `BINARY-xxx` usages as well) perform identically, within the measurement tolerances of the test.

Imagine my surprise, however, when I discovered that the use of `-fnotrunc` switch also made a difference:

```
Test 4:
USAGE DISPLAY:   1.72 SECONDS
USAGE COMP:      0.07 SECONDS
USAGE BINARY:    0.02 SECONDS

Test 5:
USAGE DISPLAY:   1.72 SECONDS
USAGE COMP:      0.07 SECONDS
USAGE BINARY:    0.02 SECONDS

Test 6:
USAGE DISPLAY:   1.73 SECONDS
USAGE COMP:      0.06 SECONDS
USAGE BINARY:    0.02 SECONDS
```

As you can see, there was a huge drop in `USAGE COMP` timings by turning off truncation. As a result, I see absolutely no reason whatsoever why the `-fnotrunc` switch option shouldn't be used on all GnuCOBOL compilations.

If you want to squeeze every last bit of performance out of your GnuCOBOL programs, don't forget to investigate the `-O` switch, `-O2` switch and the `-Os` switch, all of which influence the optimization of compiled code. Actually run programs using various optimization switches (or not) and compare execution times against those of unoptimized compiled versions of your programs. Don't just compare the generated C code because sometimes the differences can't be seen at the C source-code level.

```
Test 7:
cobc -x demomath.cbl -O2;demomath
USAGE DISPLAY:   1.68 SECONDS
USAGE COMP:      0.60 SECONDS
USAGE BINARY:    0.00 SECONDS

Test 8:
cobc -x demomath.cbl -fnotrunc -O2;demomath
USAGE DISPLAY:   1.67 SECONDS
USAGE COMP:      0.01 SECONDS
USAGE BINARY:    0.00 SECONDS
```

All tests above carried out under Linux with a AMD FX8350 under very low loading prior to the test. I would have also tried on a i7-7700 but that is under Windows 10 and I do not have a GC version on it - Vince.

End of Chapter 10 — Interfacing With The OS

11. Sub-Programming

11.1. Subprogram Types

Simply stated, a *Subprogram* is a program that is invoked by another program; the subprogram performs whatever its designed operations are and — when complete — typically returns control back to the program that invoked it. There are two different types of subprograms supported by GnuCOBOL, subroutines and user-defined functions. The distinction between these two subprogram types lies in the manner in which they are executed.

When program *A* invokes subprogram *B* as a *Subroutine*, it does so using a special statement dedicated to that function (the `CALL` statement (see [CALL], page 293), just as if *B* were one of the built-in system subroutines.

When program *A* invokes program *B* as a *User-Defined Function*, it does so in a manner identical to how *B* would have been invoked had it been one of the many built-in intrinsic functions.

In either instance, program *A* is referred to as the *Calling Program* while program *B* is known as the *Called Program*. GnuCOBOL programs may be a calling program, a called program or both.

A program written in the C programming language may serve as either the calling or called program too. A called program may act as a calling program to another called program. When a calling program does not serve as a called program to any program, that calling program is known as a *Main Program*.

Both subroutines and user-defined functions may return a value. The value they return must be an integer in the range -2147483648 to +2147483647. This value will be available in the `RETURN-CODE` special register (see [Special Registers], page 265) and also as the value of the data item specified on the `RETURNING` (see [CALL], page 293) clause of a subroutine's `CALL`.

11.2. Independent vs Contained vs Nested Subprograms

Subprograms (either subroutines or user-defined functions) can be implemented in three different ways.

Independent Subprograms
: These are subprograms that are coded as the only COBOL program in their Compilation Unit (see [Compilation Unit], page 640).

Contained Subprograms
: These are subprograms which occur in the same Compilation Unit as a main program and/or other subprograms. Each contained subprogram is separated from the next via an `END PROGRAM` marker line. As an example. . .

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUB1.
...
END PROGRAM SUB1.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUB2.
...
END PROGRAM SUB2.
```

Program source code may be concatenated as shown here, provided an `END PROGRAM` marker naming the `PROGRAM-ID` of the just-completed program is used to separate one program from another.

There's no reason that user-defined functions cannot be included too — they'll just have `FUNCTION-ID`s and will be ended by `END FUNCTION` markers.

The last program in any GnuCOBOL source file need not have an `END` marker.

When multiple programs occur in a source file, it is assumed that the programs are related to one another in that they will be `CALL`ed or executed as functions from the others.

Nested Subprograms

It is also possible to create source files where GnuCOBOL programs are nested inside each other. Take for example these four GnuCOBOL programs:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG1.
...
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG2.
...
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG3.
...
END PROGRAM PROG3.
END PROGRAM PROG2.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG4.
...
END PROGRAM PROG4.
END PROGRAM PROG1.
```

Here we see that `PROG2` is nested inside of `PROG1` because there is no `END PROGRAM` marker separating them. This means that data items or files defined within `PROG1` can be used within `PROG2` simply by attaching the `GLOBAL` (see [GLOBAL], page 176) attribute to them back in `PROG1` when they are defined.

Similarly, since there is no `END PROGRAM` marker separating `PROG3` from `PROG2`, it is possible for `PROG3` to access `GLOBAL` files and data items defined within `PROG2`. Since `PROG2` is nested within `PROG1`, any `GLOBAL` resources defined within `PROG1` will be available to `PROG3` as well.

The two `END PROGRAM` markers for `PROG3` and `PROG2` (note their sequence) mean that PROG4 is nested within `PROG1` only. It will not have access to any `GLOBAL` resources defined within either `PROG2` or `PROG3`.

The `END PROGRAM PROG1.` marker, since it is the last line in the source file, is entirely optional.

11.3. Alternate Entry Points

Any subroutine may have multiple entry-points defined within it. This means the subroutine could be called either via a `CALL 'program-id'` or a `CALL 'entry-point'` statement. There may be any number of alternate entry-points defined within a subroutine.

Alternate entry-points provide multiple ways in which the same subroutine may be called; presumably, each entry-point will provide some different functionality to the calling program. For example, if you wished to write a subroutine that manipulates "student" records in a database, you might have the primary entry-point name retrieve a student record from the database, while the alternate entry points `Add-Student`, `Update-Student` and `Delete-Student` could provide the alternate functions implied by their entry-point names.

The alternative to using multiple entry points in your subroutine, by the way, would be to include an additional argument to the primary (and only) entry point of the subroutine; this new argument might be named `STUDENT-FUNCTION` and might have values of '`FETCH`', '`ADD`', '`UPDATE`' or '`DELETE`'.

The primary entry-point for any subroutine is always the first executable statement following any `DECLARATIVES` (see [DECLARATIVES], page 254) in the procedure division. The name of that entry-point (the name that will be called) is the subroutine's `PROGRAM-ID` (see [IDENTIFICATION DIVISION], page 85).

An alternate entry point is added to a subroutine using the `ENTRY` statement (see [ENTRY], page 318).

When an alternate entry-point is called, execution within the subroutine will begin at the first executable statement following the `ENTRY` statement.

11.4. Dynamic vs Static Subprograms

Any subprogram may be either statically or dynamically loaded into memory.

A *Static Subprogram* is one which was in the same Compilation Unit (see [Compilation Unit], page 640) as the other program(s) which call it, therefore meaning that its executable object code is part of the same executable file as its calling program. The static subprogram was therefore loaded into memory as part of and at the same time as the calling program.

A *Dynamic Subprogram* is one whose executable object code exists as an executable file separate from that containing the calling program; these two programs were therefore each compiled in their own separate Compilation Group (see [Compilation Group], page 640). Dynamic subprograms are located and loaded into memory the first time they are executed. Dynamic subprograms may be unloaded from memory via the `CANCEL` statement (see [CANCEL], page 298), if desired.

GnuCOBOL subprograms may be created as either static or dynamic subprograms, as desired by the programmer.

To demonstrate, assume that a GnuCOBOL Main Program (whose code resides in the file `M.cbl`) will be calling three subprograms, named *A*, *B* and *C* (these are the `PROGRAM-ID`s

of the three subprograms, and their source code may be found in the files `A.cbl`, `B.cbl` and `C.cbl`, respectively.

Here is how these four programs would be compiled if the three subprograms are to be static:

```
cobc -x M.cbl A.cbl B.cbl C.cbl
```

This command informs the compiler (cobc) that four programs are to be compiled (the first named on the command must always be the main program), and a single executable file is to be created (due to the `-x` switch).

Here is how the main program and the three subprograms could be compiled if the three subprograms are to be dynamic:

```
cobc -x M.cbl
cobc -m A.cbl B.cbl C.cbl
```

These commands will create an executable file for the main program (`-x` switch) and three separate dynamically-loadable libraries (see `-m` switch), one for each of the three subprograms. Had we wished, we could have created a single dynamically-loadable library containing all three subprograms by adding the `-b` switch to their compilation:

```
cobc -m -b A.cbl B.cbl C.cbl
```

Dynamically-loadable *libraries* are also known by the term dynamically-loadable *modules*. The two terms are synonymous.

Here are the rules about GnuCOBOL dynamically-loadable modules:

1. There may be multiple GnuCOBOL subprograms contained within a single dynamically-loadable library if the `-b` switch is used in addition to `-m`. If not, each subprogram will be compiled to a separate dynamically-loadable library.
2. Dynamically-loadable modules will be named `xxxxxxxx.dll` on a Windows system, `xxxxxxxx.so` on a Unix system or `xxxxxxxx.dylib` on an OSX system, where `xxxxxxxx` exactly matches, including the usage of upper- and lower-case letters, the primary entry-point name (`PROGRAM-ID` or `FUNCTION-ID`) or an alternate entry point name defined via the `ENTRY` statement (see [ENTRY], page 318) of any one of the GnuCOBOL programs included in that module.
3. The first time any of the GnuCOBOL subprograms in a dynamically-loadable module are invoked, the entry-point referenced must be the one for which the `.dll`, `.so` or `.dylib` file is named.
4. When a dynamically-loadable module needs to be loaded (because it is not already in memory from a previous subprogram execution), the dynamically-loadable library will be sought in the same directory from which the main program was loaded. If it cannot be found there, each directory named in the `PATH` run-time environment variable (see [Run Time Environment Variables], page 658) will be searched. If it was not located in any of those directories, the library specified by the `COB_LIBRARY_PATH` run-time environment variable will be searched. Finally, if it *still* cannot be located, execution will be terminated with an error message (*libcob: Cannot find module 'xxxxxxx'*).
5. Once the dynamically-loadable module has been successfully loaded, any of the entry-points contained within it are now available for reference.

6. Dynamically-loadable modules may be removed from memory via the `CANCEL` statement (see [CANCEL], page 298).
7. Once a dynamically-loadable module is actually loaded into memory, even if it is subsequently unloaded (via the `CANCEL` statement), its list of entry-points remain available to the GnuCOBOL run-time library and subsequent re-executions of any of those entry points will be able to bypass the search (rule #4) as well as the *first-execution rule* (rule #3).

Consult the documentation on the `COB_PRE_LOAD` run-time environment variable, `COB_PHYSICAL_CANCEL` run-time environment variable and `COB_LOAD_CASE` run-time environment variable run-time environment variables (see [Run Time Environment Variables], page 658) for additional options when using dynamically-loadable modules.

11.5. Subprogram Execution Flow

When a subprogram is invoked, the flow of execution will differ slightly depending on whether the subprogram is a subroutine or a user-defined function.

11.5.1. Subroutine Execution Flow

When a subroutine is `CALL`ed:

1. The calling program issues a statement of the form `CALL 'entry-point' USING ...` to transfer control to the subroutine.
2. The executable for the called program will be located and loaded into memory:
   A. If it is a static subroutine, it will already be part of the executable program issuing the `CALL` (see [CALL], page 293).
   B. If it is a dynamic subroutine, the GnuCOBOL run-time system will check to see if a dynamically-loadable module containing the subprogram's entry point was already located. If it was, no further "location" activity is needed. If not, the dynamically-loadable module will be located (see [Locating Dynamically-Loadable Modules], page 682).
   C. Once the module has been located (if location was needed), it will be loaded into memory (if not already loaded).
3. Execution of the calling program is suspended and control will transfer to the called program, as follows:
   A. If the `PROGRAM-ID` (see [IDENTIFICATION DIVISION], page 85) clause of the subprogram included the `INITIAL` clause, the program will be reinitialized back to its compile-time state. This will happen regardless of the `INITIAL` clause the first time the subprogram is executed.
   B. Local-storage, if any, will be allocated and initialized.
   C. Execution will begin at the first executable statement following the subprograms entry-point. The entry point will be either the first executable statement following any `DECLARATIVES` (see [DECLARATIVES], page 254) that might be present (if the subprogram was invoked using its primary entry-point name) or the first executable statement following the `ENTRY` statement (see [ENTRY], page 318) naming the

entry-point specified on the `CALL` if the subprogram was invoked using an alternate entry point.

4. The flow of execution will then progress through the coding of the subprogram as it would with any other program.
5. If the subprogram issues a `STOP` statement (see [STOP], page 399) with the `RUN` option, program execution ceases and control returns to the operating system or whatever execution shell invoked the main program.
6. If the subprogram wishes to return control back to the calling program, it will do so using either the `GOBACK` statement (see [GOBACK], page 334) or the `EXIT PROGRAM` statement (see [EXIT], page 328). At this time:
   A. If the subprograms procedure division header or `ENTRY` statement included a `RETURNING`, the value of the data item found on that clause is moved to the `RETURN-CODE` special register (see [Special Registers], page 265); this behaviour can be altered utilizing the `CALL-CONVENTION` (see [SPECIAL-NAMES], page 92) feature to leave `RETURN-CODE` unchanged.
   B. Local-storage, if any, is de-allocated.
   C. If the calling program included a `RETURNING` clause on the `CALL` statement that invoked the subprogram, the value of the `RETURNING` data item in the subroutine is moved to that data item. If there was no `RETURNING` specified in the subroutine, the value of the `RETURN-CODE` special register is moved to that data item.
   D. Execution will resume back in the calling program with the first executable statement following the `CALL` that invoked the subprogram.

11.5.2. User-Defined Function Execution Flow

When a user-defined function is executed:

1. The object code for the called program (the user-defined function) will be located, as follows:
   A. If it is a static user-defined function, it will already be part of the executable file containing the calling program.
   B. If it is a dynamic user-defined function, the GnuCOBOL run-time system will check to see if a dynamically-loadable module containing the function's entry point was already located. If it was, no further "location" activity is needed. If not, the dynamically-loadable module will be located (see [Locating Dynamically-Loadable Modules], page 682).
   C. Once the module has been located (if location was needed), it will be loaded into memory (if not already loaded).
2. Execution of the calling program is suspended and control will transfer to the called program, as follows:
   A. Local-storage, if any, will be allocated and initialized.
   B. Execution will begin with the first executable statement in the procedure division following any `DECLARATIVES` (see [DECLARATIVES], page 254) that might be present.

3. The flow of execution will then progress through the coding of the function as it would with any other program.
4. If the function issues a `STOP` statement (see [STOP], page 399) with the `RUN` option, program execution ceases and control returns to the operating system or whatever execution shell invoked the main program.
5. If the function wishes to return control back to the calling program, it will do so using either the `GOBACK` statement (see [GOBACK], page 334) or the `EXIT FUNCTION` statement (see [EXIT], page 328). At this time:
   A. The value of the data item found on the user-defined functions `PROCEDURE DIVISION RETURNING` (see [PROCEDURE DIVISION RETURNING], page 252) clause is moved to the `RETURN-CODE` special register (see [Special Registers], page 265).
   B. Local-storage, if any, is de-allocated.
   C. Execution will resume back in the calling program at the point where the returned value of the function is needed. At that point, the value in the `RETURN-CODE` special register will be used for the function's value.

11.6. Sharing Data Between Calling and Called Programs

11.6.1. Subprogram Arguments

11.6.1.1. Calling Program Considerations

Data items defined in a calling program may be passed to either type of called program (subroutine or user-defined function) as arguments.

Arguments must be described in both the calling and called programs, and while they don't need to have the same names in both programs, they should be described in an identical manner with regard to the following characteristics:

- `PICTURE` (see [PICTURE], page 194) (including both type and length)
- `SIGN` (see [SIGN], page 513)
- `SYNCRONIZED` (see [SYNCRONIZED], page 218)
- `USAGE` (see [USAGE], page 232)

A subroutine may be passed a maximum of 251 arguments; if you build the GnuCOBOL software yourself from the distributed source, you CAN change this value by altering the defined value of `COB_MAX_FIELD_PARAMS` in the `call.h` header file but also see 7.8.5.11 for more information. There is no built-in GnuCOBOL limit to how many arguments a user-defined function may be passed.

Whether or not changes made to an argument within a subroutine will be "visible" to the calling program depends on how the argument was passed. There are three ways in which arguments may be passed from a calling program to a subroutine, as defined by the use of optional `BY` clauses in the `CALL` (see [CALL], page 293) statement's list of arguments.

As an example, the following statement passes three arguments to a subroutine — each argument is passed differently.

```
CALL "subroutine" USING BY REFERENCE arg-1
                        BY CONTENT arg-2
                        BY VALUE arg-3
END-CALL
```

The three ways arguments are passed are as follows.

`BY REFERENCE`

When a subroutine argument is passed `BY REFERENCE`, the subroutine is passed the *address* of the *actual data item* being passed as an argument. The item may be anything defined within the data division of the program. If the subroutine modifies the contents of this argument, the calling program will "see" the results of that change when the subroutine returns control. This is the default manner in which GnuCOBOL passes arguments to a subroutine, should no `BY` clauses be included on the `CALL`.

`BY CONTENT`

When a subroutine is passed an argument `BY CONTENT`, the subroutine is passed the *address* of a *copy* of the actual data being passed as an argument. The item may be anything defined within the data division of the program. The copy is

made each time the CALL statement is executed, immediately before the CALL actually takes place. If the subroutine modifies the contents of this argument, it will be the *copy* that is modified, not the original data item; the calling program will therefore not "see" the results of that change when the subroutine returns control.

BY VALUE

Passing a subroutine argument BY VALUE passes the *actual value* of the data being passed as an argument. The item may be any elementary binary numeric item defined within the data division of the program. If the subroutine modifies the contents of this argument, the calling program will not "see" the results of that change when the subroutine returns control.

The first two ways in which arguments may be passed (BY REFERENCE and BY CONTENT) are intended for use when a GnuCOBOL program is being called, while the first and third (BY REFERENCE and BY VALUE) are intended for use when a C program is being called. You *can* use BY VALUE arguments when calling GnuCOBOL subroutines, but remember that those arguments are limited to being a numeric binary data item.

Arguments to user-defined functions are always passed BY REFERENCE.

11.6.1.2. Called Program Considerations

When coding a GnuCOBOL subprogram (a subroutine or user-defined function), all arguments to the subprogram must be defined in the subprogram's linkage section.

These arguments must be explicitly included on the PROCEDURE DIVISION USING (see [PROCEDURE DIVISION USING], page 248) clause that lists the arguments in the sequence in which they will be passed to the subprogram.

These arguments described in the PROCEDURE DIVISION USING clause may each be defined as either BY REFERENCE, if the calling program is passing them either BY REFERENCE or BY CONTENT, or as BY VALUE if they are being passed BY VALUE.

By default, all arguments are assumed to be BY REFERENCE unless explicitly stated otherwise on the procedure division header.

Arguments to a user-defined function are always to be specified as BY REFERENCE (either explicitly or by not using any BY).

If the subprogram returns a value, the data item in which the value is returned must also be defined in the subprogram's linkage section, with a USAGE (see [USAGE], page 232) of BINARY-LONG SIGNED, or its equivalent.

11.6.2. GLOBAL Data Items

Another way in which a data item may be shared between a calling program (A) and a called program (B) is by defining the data item in the calling program and attaching the GLOBAL (see [GLOBAL], page 176) clause to it so that it may be used within the called program. In order for this to work, program B (the one called by program A) must be a nested subprogram within program A.

Here's a small example:

```
IDENTIFICATION DIVISION.
```

```
PROGRAM-ID. DemoGLOBAL.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Arg GLOBAL                     PIC X(10).
PROCEDURE DIVISION.
000-Main.
    MOVE ALL "X" TO Arg
    CALL "DemoSub" END-CALL
    DISPLAY "DemoGLOBAL: " Arg END-DISPLAY
    GOBACK
    .
IDENTIFICATION DIVISION.
PROGRAM-ID. DemoSub.
PROCEDURE DIVISION.
000-Main.
    MOVE ALL "*" TO Arg.
    GOBACK
    .
END PROGRAM DemoSub.
END PROGRAM DemoGLOBAL.
```

When the program runs, it produces the output:

```
DemoGLOBAL: **********
```

11.6.3. EXTERNAL Data Items

The final way in which a data item may be shared between a calling program (A) and a called program (B) is by defining the data item (with the same name) in both programs and attaching the `EXTERNAL` (see [EXTERNAL], page 171) clause to it (again, in both programs). This approach works regardless of whether the called program is nested within the calling program or not. It also works even if the two programs are compiled separately.

Here's a demonstration:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DemoEXTERNAL.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Arg EXTERNAL                 PIC X(10).
PROCEDURE DIVISION.
000-Main.
    MOVE ALL "X" TO Arg
    CALL "DemoSub" END-CALL
    DISPLAY "DemoEXTERNAL: " Arg END-DISPLAY
    GOBACK
    .
END PROGRAM DemoEXTERNAL.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DemoSub.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Arg EXTERNAL                PIC X(10).
PROCEDURE DIVISION.
000-Main.
    MOVE ALL "*" TO Arg.
    GOBACK
    .
END PROGRAM DemoSub.
```

When the program runs, it produces the output:

```
DemoEXTERNAL: **********
```

11.7. Recursive Subprograms

A subroutine may `CALL` itself, either directly or indirectly from another subroutine or user-defined function that it `CALL`s. Any subroutine that indulges in this sort of behaviour (called recursion) is called a *Recursive Subprogram*.

Any GnuCOBOL subroutine can be recursively invoked only if it is defined to the GnuCOBOL compiler as *being* a recursive subroutine. This is accomplished by adding the `RECURSIVE` attribute to its `PROGRAM-ID` (see [IDENTIFICATION DIVISION], page 85).

All User-defined functions are automatically capable of being executed recursively.

Here is an example of a main program (DEMOFACT) that calls both a subprogram (SUB) and a user-defined function (FUNC) to compute the factorial value of a number.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DEMOFACT.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
    FUNCTION RECURSIVEFUNC.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Result    USAGE BINARY-LONG.
01  Arg       USAGE BINARY-LONG.
PROCEDURE DIVISION.
000-Main.
    MOVE 6 TO Arg
    CALL "RECURSIVESUB"
        USING BY CONTENT Arg
        RETURNING Result
    DISPLAY Arg "! = "
            Result
    DISPLAY Arg "! = "
            RECURSIVEFUNC(Arg)
```

```
    GOBACK
    .
END PROGRAM DEMOFACT.
IDENTIFICATION DIVISION.
PROGRAM-ID. SUB RECURSIVE.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Result      USAGE BINARY-LONG.
01  Next-Arg    USAGE BINARY-LONG.
01  Next-Result USAGE BINARY-LONG.
LINKAGE SECTION.
01  Arg         USAGE BINARY-LONG.
PROCEDURE DIVISION USING Arg
               RETURNING Result.
000-Main.
    DISPLAY "Entering SUB"
            " Arg=" Arg
    IF Arg = 1
      MOVE 1 TO Result
      DISPLAY "Leaving SUB"
              " Returning " Result
    ELSE
      SUBTRACT 1 FROM Arg
          GIVING Next-Arg
      CALL "SUB"
           USING BY CONTENT Next-Arg
           RETURNING Next-Result
      COMPUTE Result =
              Arg * Next-Result
      DISPLAY "Leaving SUB"
              " Returning "
              Result "=" Arg "*"
              Next-Result
    END-IF
    GOBACK
    .
END PROGRAM SUB.
IDENTIFICATION DIVISION.
FUNCTION-ID. FUNC.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
    FUNCTION RECURSIVEFUNC.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
```

```
01  Arg     USAGE BINARY-LONG.
01  Result  USAGE BINARY-LONG
             SIGNED.
PROCEDURE DIVISION USING Arg
                RETURNING Result.
000-Main.
    DISPLAY "Entering FUNC"
            " Arg=" Arg
    IF Arg = 1
      MOVE 1 TO Result
    ELSE
      COMPUTE Result = Arg *
              FUNC(Arg - 1)
    END-IF
    DISPLAY "Leaving FUNC"
            " Returning " Result
    GOBACK
    .
END FUNCTION FUNC.
```

When DEMOFACT is executed, the output shown below is generated.

```
E:\Programs\Demos>demofact
Entering RECURSIVESUB Arg=+0000000006
Entering RECURSIVESUB Arg=+0000000005
Entering RECURSIVESUB Arg=+0000000004
Entering RECURSIVESUB Arg=+0000000003
Entering RECURSIVESUB Arg=+0000000002
Entering RECURSIVESUB Arg=+0000000001
Leaving RECURSIVESUB Returning +0000000001
Leaving RECURSIVESUB Returning +0000000002=+0000000002*+0000000001
Leaving RECURSIVESUB Returning +0000000006=+0000000003*+0000000002
Leaving RECURSIVESUB Returning +0000000024=+0000000004*+0000000006
Leaving RECURSIVESUB Returning +0000000120=+0000000005*+0000000024
Leaving RECURSIVESUB Returning +0000000720=+0000000006*+0000000120
+0000000006! = +0000000720
Entering RECURSIVEFUNC Arg=+0000000006
Entering RECURSIVEFUNC Arg=+0000000005
Entering RECURSIVEFUNC Arg=+0000000004
Entering RECURSIVEFUNC Arg=+0000000003
Entering RECURSIVEFUNC Arg=+0000000002
Entering RECURSIVEFUNC Arg=+0000000001
Leaving RECURSIVEFUNC Returning +0000000001
Leaving RECURSIVEFUNC Returning +0000000002
Leaving RECURSIVEFUNC Returning +0000000006
Leaving RECURSIVEFUNC Returning +0000000024
Leaving RECURSIVEFUNC Returning +0000000120
```

```
Leaving RECURSIVEFUNC Returning +0000000720
+0000000006! = +0000000720
```

11.8. Combining GnuCOBOL and C Programs

The upcoming sections deal the issues pertaining to calling C language programs from GnuCOBOL programs, and vice versa. Two additional sections provide samples illustrating specifics as to how those issues are overcome in actual program code.

11.8.1. GnuCOBOL Run-Time Library Requirements

Like most other implementations of the COBOL language, GnuCOBOL utilizes a run-time library. When the first program executed in a given execution sequence is a GnuCOBOL program, any run-time library initialization will be performed by the compiled COBOL code in a manner that is transparent to the C-language programmer. If, however, a C program is the first to execute, the burden of performing GnuCOBOL run-time library initialization falls upon the C program. See [C Main Programs Calling GnuCOBOL Subprograms], page 696, for an example of how to do this.

11.8.2. String Allocation Differences Between GnuCOBOL and C

Both languages store strings as a fixed-length continuous sequence of characters.

COBOL stores these character sequences up to a specific quantity limit imposed by the `PICTURE` (see [PICTURE], page 194) clause of the data item. For example: `01 LastName PIC X(15).`.

There is never an issue of exactly what the length of a string contained in a `USAGE DISPLAY` (see [USAGE], page 232) data item is — there are always exactly how ever many characters as were allowed for by the `PICTURE` clause. In the example above, `LastName` will always contain exactly fifteen characters; of course, there may be anywhere from 0 to 15 trailing SPACES as part of the current LastName value.

C actually has no "string" data type; it stores strings as an array of `char` data type items where each element of the array is a single character. Being an array, there is an upper limit to how many characters may be stored in a given "string". For example:

```
char lastName[15]; /* 15 chars: lastName[0] through lastName[14] */
```

C provides a robust set of string-manipulation functions to copy strings from one char array to another, search strings for certain characters, compare one char array to another, concatenate char arrays and so forth. To make these functions possible, it was necessary to be able to define the logical end of a string. C accomplishes this via the expectation that all strings (char arrays) will be terminated by a NULL character (`x'00'`). Of course, no one forces a programmer to do this, but if [s]he ever expects to use any of the C standard functions to manipulate that string they had *better* be null-terminating their strings!

So, GnuCOBOL programmers expecting to pass strings to or receive strings from C programs had best be prepared to deal with the null-termination issue, as follows:

1. Pass a quoted literal string from GnuCOBOL to C as a zero-delimited string literal (`Z'string'`).

2. Pass alphanumeric (`PIC X`) or alphabetic (`PIC A`) data items to C subroutines by appending an ASCII `NUL` character (`X'00'`) to them. For example, to pass the 15-character `LastName` data item described above to a C subroutine:

```
01  LastName-Arg-to-C     PIC X(16).
...
    MOVE FUNCTION CONCATENATE(LastName,X'00') TO LastName-Arg-to-C
```

   And then pass `LastName-Arg-to-C` to the C subprogram!

3. When a COBOL program needs to process string data prepared by a C program, the embedded null character must be accounted for. This can easily be accomplished with an `INSPECT` statement (see [INSPECT], page 344) such as the following:

```
INSPECT Data-From-a-C-Program
    REPLACING FIRST X'00' BY SPACE
              CHARACTERS BY SPACE AFTER INITIAL X'00'
```

11.8.3. Matching C Data Types with GnuCOBOL USAGE's

Matching up GnuCOBOL numeric Usage's with their C language data type equivalents is possible via the following chart:

COBOL	C
BINARY-CHAR UNSIGNED	unsigned char
BINARY-CHAR [SIGNED]	signed char
BINARY-SHORT UNSIGNED	unsigned
	unsigned int
	unsigned short
	unsigned short int
BINARY-SHORT [SIGNED]	int
	short
	short int
	signed int
	signed short
	signed short int
BINARY-LONG UNSIGNED	unsigned long
	unsigned long int
BINARY-LONG [SIGNED]	long
BINARY-INT	long int
	signed long
	signed long int
BINARY-C-LONG [SIGNED]	long
BINARY-DOUBLE UNSIGNED	unsigned long long
	unsigned long long int
BINARY-DOUBLE [SIGNED]	long long int
BINARY-LONG-LONG	signed long long int
COMPUTATIONAL-1	float
COMPUTATIONAL-2	double
N/A (no GnuCOBOL equivalent)	long double

These sizes conform to the COBOL standard and the minimum sizes of the COBOL types are the same as the minimum sizes of the corresponding C data types. There's no official compatibility between them. Note that values in square braces '[]' are the defaults.

11.8.4. GnuCOBOL Main Programs CALLing C Subprograms

Here's a sample of a GnuCOBOL program that CALLs a C subprogram.

```
COBOL Calling Program               C Called Program
================================    ==============================
IDENTIFICATION DIVISION.            #include <stdio.h>
PROGRAM-ID. maincob.                int subc(char *arg1,
DATA DIVISION.                               char *arg2,
WORKING-STORAGE SECTION.                     unsigned long *arg3) {
01  Arg1     PIC X(7).               char nu1[7]="New1";
01  Arg2     PIC X(7).               char nu2[7]="New2";
01  Arg3     USAGE BINARY-LONG.      printf("Starting subc\n");
PROCEDURE DIVISION.                 printf("Arg1=%s\n",arg1);
000-Main.                           printf("Arg2=%s\n",arg2);
    DISPLAY 'Starting maincob'      printf("Arg3=%d\n",*arg3);
```

```
MOVE Z'Arg1'   TO Arg1            arg1[0]='X';
MOVE Z'Arg2'   TO Arg2            arg2[0]='Y';
MOVE 123456789 TO Arg3            *arg3=987654321;
CALL 'subc'                       return 2;
    USING BY CONTENT   Arg1,    }
          BY REFERENCE Arg2,
          BY REFERENCE Arg3
DISPLAY 'Back'
DISPLAY 'Arg1=' Arg1
DISPLAY 'Arg2=' Arg2
DISPLAY 'Arg3=' Arg3
DISPLAY 'Returned value='
        RETURN-CODE
STOP RUN
.
```

The idea is to pass two string and one full-word unsigned arguments to the subprogram, have the subprogram print them out, change all three and pass a return code of 2 back to the caller. The caller will then re-display the three arguments (showing changes only to the two `BY REFERENCE` arguments), display the return code and halt.

While simple, these two programs illustrate the techniques required quite nicely.

Note how the COBOL program ensures that a null end-of-string terminator is present on both string arguments.

Since the C program is planning on making changes to all three arguments, it declares all three as pointers in the function header and references the third argument as a pointer in the function body. It actually had no choice for the two string (char array) arguments — they must be defined as pointers in the function even though the function code references them without the leading '`*`' that normally signifies pointers.

These programs are compiled and executed as follows.

```
$ cobc -x maincob.cbl subc.c
$ maincob
Starting maincob
Starting subc
Arg1=Arg1
Arg2=Arg2
Arg3=123456789
Back
Arg1=Arg1
Arg2=Yrg2
Arg3=+0987654321
Returned value=+000000002
$
```

Remember that the null characters are actually in the GnuCOBOL `Arg1` and `Arg2` data items. They don't appear in the output, but they *are* there.

Did you notice the output showing the contents of `Arg1` after the subroutine was called? Those contents were unchanged! The subroutine *definitely* changed that argument, but since the COBOL program passed that argument `BY CONTENT`, the change was made to a *copy* of the argument, not to the `Arg1` data item itself.

11.8.5. C Main Programs Calling GnuCOBOL Subprograms

Now, the roles of the two languages in the previous section will be reversed, having a C main program execute a GnuCOBOL subprogram.

```
C Calling Program
============================================
#include <libcob.h> /* COB RUN-TIME */
#include <stdio.h>
int main (int argc, char **argv) {
   int returnCode;
   char arg1[7] = "Arg1";
   char arg2[7] = "Arg2";
   unsigned long arg3 = 123456789;
   printf("Starting mainc...\n");
   cob_init (argc, argv); /* COB RUN-TIME */
   returnCode = subcob(arg1,arg2,&arg3);
   printf("Back\n");
   printf("Arg1=%s\n",arg1);
   printf("Arg2=%s\n",arg2);
   printf("Arg3=%d\n",arg3);
   printf("Returned value=%d\n",returnCode);
   return returnCode;
}
```

```
GNU-COBOL Called Program
================================
IDENTIFICATION DIVISION.
PROGRAM-ID. subcob.
DATA DIVISION.
LINKAGE SECTION.
01  Arg1      PIC X(7).
01  Arg2      PIC X(7).
01  Arg3      USAGE BINARY-LONG.
PROCEDURE DIVISION USING
    BY VALUE     Arg1,
    BY REFERENCE Arg2,
    BY REFERENCE Arg3.
000-Main.
    DISPLAY 'Starting cobsub.cbl'
    DISPLAY 'Arg1=' Arg1
    DISPLAY 'Arg2=' Arg2
    DISPLAY 'Arg3=' Arg3
    MOVE 'X' TO Arg1 (1:1)
    MOVE 'Y' TO Arg2 (1:1)
    MOVE 987654321 TO Arg3
    MOVE 2 TO RETURN-CODE
    GOBACK
    .
```

Since the C program is the one that will execute first, before the GnuCOBOL subroutine, the burden of initializing the GnuCOBOL run-time environment lies with that C program; it will have to invoke the `cob_init` function, which is part of the `libcob` library. The two required C statements are shown highlighted.

The arguments to the `cob_init` routine are the argument count and value parameters passed to the main function when the program began execution. By passing them into the GnuCOBOL subprogram, it will be possible for that GnuCOBOL program to retrieve the command line or individual command-line arguments. If that won't be necessary, `cob_init(0,NULL);` could be specified instead.

Since the C program wants to allow `arg3` to be changed by the subprogram, it prefixes it with a '`&`' to force a `CALL BY REFERENCE` for that argument. Since `arg1` and `arg2` are strings (char arrays), they are automatically passed by reference.

Here's the output of the compilation process as well as the program's execution. The example assumes a Windows system with a GnuCOBOL build that uses the GNU C compiler on that system; the technique works equally well regardless of which C compiler and which operating system you're using.

```
C:\Users\Gary\Documents\Programs> cobc -S subcob.cbl
C:\Users\Gary\Documents\Programs> gcc mainc.c subcob.s -o mainc.exe -llibcob
C:\Users\Gary\Documents\Programs> mainc.exe
Starting mainc...
Starting cobsub.cbl
Arg1=Arg1
```

```
Arg2=Arg2
Arg3=+0123456789
Back
Arg1=Xrg1
Arg2=Yrg2
Arg3=987654321
Returned value=2
C:\Users\Gary\Documents\Programs>
```

Note that even though we told GnuCOBOL that the 1st argument was to be `BY VALUE`, it was treated as if it were `BY REFERENCE` anyway. String (char array) arguments passed from C callers to GnuCOBOL subprograms will be modifiable by the subprogram. It's best to pass a copy of such data if you want to ensure that the subprogram doesn't change it.

The third argument is different, however. Since it's not an array you have the choice of passing it either `BY REFERENCE` or `BY VALUE`.

End of Chapter 11 — Sub-Programming

12. Programming Style Suggestions

This chapter deals with a variety of stylistic issues that may be of interest to someone who is just starting out learning and using COBOL. Much of this chapter makes recommendations and suggestions for how to write your own programs. The sample programs in the *Sample Programs* document (see *Sample Programs*) were coded using almost all of these recommendations.

There's no particular order of importance to the topics presented here.

12.1. Marking Changes in Programs

Historically in the early 60's programs were first punched on to paper tape and by the mid 60's that was replaced almost totally, by punched cards although paper tape was still used by programmers for the odd few changes to their sources held on magnetic tape or disk as a portable paper tape punch could be put in your pocket. Now the problem with punched cards were there was 2,000 cards per box and that they could and did, get dropped. So, cc (column) 1 through 6 had the card sequence number in and that way if a box was dropped they could be feed in to a card sorter to be fixed. This was after the cards was cleaned up so that they were all in the same direction which one corner cut out helped.

In the late 70's cards was also on its way out to the point where P.C's started being used (and no they were not made by IBM), so these columns could be used for other purposes including cc 73 - 80 instead of indicating the 8 character program name which was the maximum size allowed on a IBM system.

For quite a while now (back to the late 1970's), the *sequence number area'* of a COBOL statement (columns 1-6) has come to be used as a change indicator area. Programmers would place a code in columns 1-6 of every line they changed in a program. The author works in a COBOL shop where change indicators of the form "xxmmyy" are required on every altered line of a program — "xx" is the initials of the programmer while "mmyy" are the month and two-digit year of the date the change was made. This is frequently accompanied by a comment block at or near the top of a COBOL program providing general documentation of what changes were made and what change indicator was used to mark that change.

The GCic sample program source listing (see Section "GCic" in *GnuCOBOL Sample Programs*) provides an excellent example of such documentation.

This technique of using columns 1-6 as a change indicator will *only* work if fixed source-record format is in effect.

Some COBOL shops prefer to use the eight-character Program Name Area (columns 73-80) as a change code area.

Marking changes becomes more of a challenge when free-format source code is in effect. Creating a top-of-program comment block to generically describe changes that have been made isn't difficult, even in free-form. What *is* difficult, however, is coming up with a scheme for per-statement mark up of changes that doesn't introduce a ridiculously excessive number of source lines to the program. I'm not sure there is a good answer to this problem (if a reader has one, please let me know). Generally, I've noticed that shops using free-format conventions for their COBOL source tend to stick with just the top-of-program comment

block combined with minimal comment blocks sprinkled throughout the program noting areas that underwent major changes.

12.2. Data Item Coding and Naming Conventions

When programs get very large, it becomes more and more challenging to keep track of the data items that will be used in the program. Here, in no particular order of importance, are a variety of conventions that can simplify that problem.

Remember that the points described here are intended to make things easier for you, the programmer. No COBOL compiler cares one way or another whether any of these suggestions are followed.

1. Avoid the use of level 77 data items in new programs. Once (1968 and before) there were valid reasons for creating level-77 data items, but since the 1974 ANSI standard of COBOL there really hasn't been any reason why an elementary level-01 data item couldn't have been used instead of a level-77 item.
2. Allocate level-01 data items in alphabetical sequence in the program source wherever practical. This will make it vastly easier to locate the definitions of 01-level items in the program source without having to resort to a compilation cross-reference listing and/or text editor "find" command to locate them.
3. *Consider* prefixing data items with an indication of where in the program structure they were created. For example:
   - Start everything defined in the file section with "F-"
   - Start everything defined in working-storage with "WS-"
   - Start everything defined in local-storage with "LS-"
   - Start everything defined in the linkage section with "L-"
   - Start everything defined in the screen section with "S-"
   - Start everything defined in the report section with "R-"

   A convention such as this makes it simple, when you're reviewing code in the procedure division, to know in which section of the data division you should look in when locating the detailed description of a data item. Once you're in the right division, coding convention #2 will assist in locating the data item definition.
4. Consider including a trailing descriptor of the nature of all data items in their names. The following chart presents a variety of such descriptors the author has encountered and used through the years.

   `-ADDR` The data item contains all or a part of an Address (`City-ADDR`, `State-ADDR`, `Street-ADDR`, ...)

   `-BOOL` A level-88 data item (which only has the value `TRUE` or `FALSE`)

   `-CD` A *code* whose value denotes information content above and beyond that of the mere value itself. Some examples could be `Error-CD`, `Status-CD`, `Billing-CD`

   `-CHR` A data item containing a single character of data.

`-CONST`	A constant, specified as a level-78 data item, a level-01 item with the `CONST` attribute
`-DT`	The data item contains a complete or partial date (`Birth-DT`, `Birth-Month-DT`, `Birth-Year-DT`, ...)
`-DTTM`	A data item containing both a date and a time
`-FILE`	A file name. Note that these items would probably also have a "F-" prefix.
`-IDX`	A data item used as a table index (see section 12.3)
`-NM`	All or a portion of a person's name. These could be extended to include business names, product names, etc.
`-PTR`	A data item whose `USAGE` is `POINTER`
`-NUM`	A generic numeric data item that doesn't fit into any of the other categories
`-QTY`	A count of something
`-REC`	An 01-level item defined in the `FILE SECTION` (constituting the layout of a record within a file). Note that these items would probably also have a "F-" prefix.
`-SCR`	The data item contains a complete or partial screen description (appropriate for SCREEN SECTION 01-level data items).
`-SUB`	A numeric item used as a table subscript (see section 12.3)
`-TEL`	All or part of a telephone number
`-TM`	The data item contains a complete or partial time value
`-TXT`	The data item contains generic alphanumeric text that doesn't fit into any of the other categories.

The above is by no means an exhaustive list, but good programmers will use as few of these descriptors as possible as having too many defeats any benefits of such classification/documentation efforts.

5. Consider including an acronym to be inserted into the name of any data item defined directly or indirectly subordinate to an 01-level item, typically to be specified after any section-level tag, if you're using them. For example, consider the names used in the following structure:

```
01  WS-File-Status-Message-TXT.
    05 FILLER                     PIC X(13) VALUE 'Status Code: '.
    05 WS-FSM-Status-CD           PIC 9(2).
    05 FILLER                     PIC X(11) VALUE ', Meaning: '.
    05 WS-FSM-Msg-TXT             PIC X(25).
....
01  WS-OI-SUB                    PIC 99  COMP.
01  WS-OI-IDX                    PIC 99  COMP.
```

The "-FSM-" acronyms make it easier to locate the description of the 01-item the status code and message text items belong to.

12.3. Table Subscripting versus Table Indexing

The elements of a table may be referenced either using a subscript or an index. Syntactically, this is coded using parenthesis, as per the following three examples, all of which store the letter '`A`' into the 17th occurrence of a data item named WS-Output-Image-TXT:

1. `MOVE 'A' TO WS-Output-Image-TXT (17)`
2. `MOVE 17 TO WS-OI-SUB`
   `MOVE 'A' TO WS-Output-Image-TXT (WS-OI-SUB)`
3. `SET WS-OI-IDX TO 17`
   `MOVE 'A' TO WS-Output-Image-TXT (WS-OI-IDX)`

The 1st and 2nd examples are referred to as *Subscripting* while the 3rd is known as *Indexing*. The distinction is fairly simple.

Indexing is the process of referencing an element of a table utilizing a data item with an explicitly or implicitly defined `USAGE` (see [USAGE], page 232) of `INDEX` to select the desired occurrence, while ...

Subscripting is the process of referencing an element of a table utilizing either a numeric constant or an unedited numeric data item to select the desired occurrence.

Various implementations of COBOL generate object code that is quite different in each of these three situations, and GnuCOBOL is no exception.

In general, table references such as example #1 (constant subscript) generate the smallest, simplest and fastest object code while table references such as example #2 (numeric data item subscript) generate the largest, most-complicated and slowest object code.

Table references such as example #3 (table indexing) generate object code that falls in the middle of the other two but is far closer in efficiency to example #1 than #2.

Some COBOL statements (`SEARCH` (see [SEARCH], page 377), `SEARCH ALL` (see [SEARCH ALL], page 379) and the table-based `SORT` (see [Table SORT], page 395)) require you to index the affected table and to utilize that index with those statements. With any other references to tables, the choice is left to the programmer as to which approach should be used. In general, follow these rules:

1. Use constant subscripts (example #1) wherever possible/practical.
2. If references to table elements are going to be performed many, many times (tens or hundreds of thousands of times or more) during program execution, you will probably see a noticeable reduction in program execution time if you use indexing versus subscripting.

It's impossible to perform any arithmetic operation against an index data item directly (other than a simple incrementation or decremental operation via the `SET UP/DOWN` statement (see [SET UP/DOWN], page 386)). Situations where any non-trivial computations are required to calculate the effective occurrence number for a table reference will require you to use a conventional unedited numeric data item as the receiving field for the calculation. That calculated value would then need to be saved into the index data item via a `SET Index` statement.

If you only need to use the computed occurrence number once, you might as well just use the computed occurrence number data item as a subscript. If, however, you will need to use

a computed "subscript" many more times than once, the run-time overhead of converting that occurrence value to an index (via `SET Index`) will be worth the coding effort.

Whew!

If references to table elements are not going to be performed many, many times it probably won't make much difference whether you use indexing or subscripting.

If you are comfortable with the C programming language, you might find the following simple GnuCOBOL program useful in exploring the differences between subscripting and indexing:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SUBVSINDEX.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-TABLE-SUB                BINARY-LONG.
01  WS-TABLE.
    05 WS-TABLE-ENTRY           OCCURS 20 TIMES
                                INDEXED BY WS-TABLE-IDX
                                PIC X(1).
PROCEDURE DIVISION.
000-Main SECTION.
E1. MOVE 'A' TO WS-TABLE-ENTRY (17)
    .
E2. MOVE 17 TO WS-TABLE-SUB
    MOVE 'A' TO WS-TABLE-ENTRY (WS-TABLE-SUB)
    .
E3. SET WS-TABLE-IDX TO 17
    MOVE 'A' TO WS-TABLE-ENTRY (WS-TABLE-SUB)
    .
```

Compile this program as follows (the assumption is made that you are executing the `cobc` command from the directory in which the above program source code (subvsindex.cbl) exists.

```
cobc -C -save-temps subvsindex.cbl\
```

After this command is executed, the file `subvsindex.c` will contain the procedure division C code and `subvsindex.c.1.h` will contain the working-storage C code. Compare the generated C code for each of the three `MOVE` statements.

12.4. Copybook Naming Conventions and Usage

Since the intent of a copybook is to introduce COBOL code into a particular spot in a program via the `COPY` statement (see [COPY], page 63), it is always a good idea to prefix copybook names with a character sequence that identifies where in a program its contents are intended to be `COPY`ed.

For example:

`IDxxxxxxxx`
: Copybooks containing code intended for the identification division. These will be rare as you almost never encounter copied code in the identification division.

`EDxxxxxxxx`
: Copybooks containing code intended for use in the environment division. These copybooks are generally used for predefined `SPECIAL-NAMES` (see [SPECIAL-NAMES], page 92) or `FILE-CONTROL` (see [INPUT-OUTPUT SECTION], page 103) syntax,

`DDxxxxxxxx`
: Copybooks that contain data definitions.

`PDxxxxxxxx`
: Copybooks that contain executable instructions.

12.5. PROCEDURE DIVISION Sections Versus Paragraphs

The issue of whether to use section and/or paragraph names (collectively referred to as procedure names) within the procedure division is one of near religious significance with many COBOL programmers.

COBOL programming standards used by many organizations that use the language generally call for procedure names to:

1. Contain a leading numeric component (for example: `2000-Update-Customer`), AND. . .
2. Be defined in the procedure division in non-decreasing sequence of that numeric component.

When you are looking at or editing any large COBOL program that has been created with programming standards that include these two rules, it is always a simple thing to know whether a reference to a procedure is being made to code that exists before or after your current location in the program, simply by comparing the numeric component of the current procedure's name with the one in question.

Technically, GnuCOBOL does not require ANY procedure names be defined unless:

1. You are using the `ALTER` statement (see [ALTER], page 292) (the use of which should be avoided at all costs)
2. You are using a procedural `PERFORM` statement (see [Procedural PERFORM], page 360)
3. You are using a `GO TO` statement (see [GO TO], page 335)
4. You are using a `MERGE` statement (see [MERGE], page 349) with an `OUTPUT PROCEDURE`
5. You are using a `SORT` statement (see [SORT], page 391) with either (or both) an `INPUT PROCEDURE` or `OUTPUT PROCEDURE`
6. You are using `DECLARATIVES` (see [DECLARATIVES], page 254)

Since it is difficult to write any non-trivial COBOL program that uses none of the above, lets assume you will be including at least one section or paragraph in your GnuCOBOL programs.

I like to use procedure division paragraphs and sections as follows:

1. The very first procedure defined in the procedure division of my programs, assuming no `DECLARATIVES` (see [DECLARATIVES], page 254) are defined, will be a section named `000-Main`. The declaration of this procedure will immediately follow the procedure division header (or `END DECLARATIVES` if `DECLARATIVES` are used).
2. Any procedures referenced by `MERGE`, `PERFORM`, or `SORT` statements will be defined as sections.
3. Any procedures referenced by `GO TO` statements will be defined as paragraphs, and those paragraphs will be defined in the same section as the `GO TO` statements that reference them. In other words, `GO TO` statements may not be used to transfer control to a point in a different section. This is *not* a GnuCOBOL rule — this is my own personal programming practice intended to improve the readability and maintainability of my programs.
4. I always include a numeric prefix to all procedure names I define, for the reasons stated earlier.
5. I do not use `THRU` on any `MERGE`, `PERFORM` or `SORT` statement unless the programming standards of the shop in which I am working require it. My reasoning for this is that it is too easy to accidentally introduce a new procedure into the scope of a `THRU`.

12.6. COMPUTE Versus ADD-SUBTRACT-MULTIPLY-DIVIDE

Over the years, there has been much debate over the efficiency and arithmetic accuracy of using the `COMPUTE` statement (see [COMPUTE], page 301) rather than the four basic arithmetic operation statements.

Here are the facts — draw your own conclusions as to which approach is more appropriate under which circumstances.

1. The `COMPUTE` statement supports exponentiation (via the '`**`' operator) — there is no equivalent basic arithmetic statement. Although you could simulate integral exponentiation (raising a value to the third power, for example) using `MULTIPLY` statements, and you may use the `SQRT` intrinsic function (see [SQRT], page 515) to find a square root, there's just no (easy) way to find the cube-root of a value without using the `COMPUTE` statement.
2. For non-trivial computations, `COMPUTE` statements "read" better. Take this, for example:

   ```
   COMPUTE R = (A + B * C) / D
   ```

   As compared to:

   ```
   MULTIPLY B BY C GIVING TEMP
   ADD A TO TEMP
   DIVIDE TEMP BY D GIVING R
   ```

   For non-trivial computations, `COMPUTE` statements may execute faster than the equivalent chain of basic arithmetic statements. For example, the COMPUTE statement shown above executes about 25% faster on my computer using GnuCOBOL than does the MULTIPLY-ADD-DIVIDE sequence.

3. For trivial computations, on the other hand, I prefer the inherent readability of a statement such as this:

   ```
   ADD 1 TO WSS-Input-Trans-QTY
   ```

   to this:

   ```
   COMPUTE WS-Input-Trans-QTY = WS-Input-Trans-QTY + 1
   ```

End of Chapter 12 — Programming Style Suggestions

13. Programming for XFD

This chapter deals with the initial support for ODBC and OCI as file-handlers (so far PostgrSQL, MySQL, SQLite, MSSQL) and OCI, along with new directory COB_SCHEMA_DIR containing the necessary internal schema files to match the file definition to the database table all though usage of $XDF.

13.1. GnuCobol use SQL for files

The ACUCOBOL compiler has a defined syntax for describing records that can be stored in SQL tables. This is referred to as XFD (extended file descriptor). The various XFD directives are intermixed in the record description of a data file as either $XFD (similar to $SET) or as comments *((XFD)). Each directive precedes the data item it is affecting. When a program is compiled that has these directives and the compiler is directed by a compile option, then the record layout is written out to a text file which is later used at run-time to process the data records. i.e., facilitate mapping the COBOL data fields to/from SQL columns.

The goal of this project is for GnuCOBOL to accept directives similar in syntax to the ACUCOBOL XFD syntax1 (plus a few additions as required) and develop run-time modules to handle all of the COBOL I/O verbs while the data is stored in a SQL table.

The purpose for doing this is to allow legacy COBOL to have data stored in SQL tables without rewriting the COBOL code. If a COBOL based application is already using EXEC SQL this feature of having INDEXED files stored in SQL tables is really not of any use. Once the data is in SQL tables it may be accessed by other SQL tools. To get access to more advanced SQL features, the COBOL code will need to be recoded to use EXEC SQL, but this can be done at points which provide the most advantage.

At present both INDEXED and RELATIVE files are supported using either an ODBC or OCI (Oracle Call Interface) interface. RELATIVE files are supported by adding a special column that gets a unique number as the RELATIVE KEY. The intention is that you could store the data for the master data files of an application in an SQL database. Often COBOL based applications have intermediate data files and it would be a waste of time to place these into the database. Also keep in mind that the performance of ISAM will be much better than using an SQL database. However, SQL provides the opportunity to use ad hoc inquiry and other third party tools to process your data.

When directed by the XFD directives, the GnuCOBOL compiler will write out the CREATE TABLE statements that could be used to represent the file. The GnuCOBOL compiler will also write out a data description file which is read at runtime and used to manage the conversion of data between the COBOL record view and the SQL column view. The format of this data description file is not exactly the same as what ACUCOBOL had used since the file is easily regenerated. The more important feature is to accept similar XFD directives intermixed in the record description.

XFD Directives

The following directives are accepted intermixed in the COBOL record description. The KEY IS clauses from SELECT statements will be used to generate SQL INDEX statements.

$XFD Name	**XFD Directive Descriptions**
ALL	Indicates this is a special module defining all files in an application system so all files in this module should be defined for use via SQL
FILE	Name to be used for SQL table, filename.ddl and filename.xd instead of guessing from the SELECT/ASSIGN. If FILE is not provided then if ASSIGN filename is used that is taken, other wise the SELECT name is used for the table name and the .ddl and .xd files.
ALPHA	Next field is handled as a single CHAR type. If a group item the entire group is a single CHAR column. May optionally have alternate name.
BINARY	Next field handled as BINARY (RAW) so arbitrary hex data is stored. If a group item the entire group becomes a single BINARY column. May optionally have alternate name
CHAR	Next field is handled as a single CHAR type. If a group item the entire group is a single CHAR column. May optionally have alternate name.
DATE	Next field is a date in the defined format M - month (01 12), Y - year (2 or 4 digit), D - day of month (01-31) H - hour (00 -23), N minute, S second, T - hundredths of a second J - Julian day, E - day of year YY%nn nn is the pivot year YYY+nnnn nnnn is added to YYY to compute the actual year
GROUP type	The next field or group item is taken as a single SQL column of the declared type
NAME	Name to be used for the next SQL column
NUMERIC	Next field is handled as NUMERIC, may optionally have alternate name
USE type	The word USE is essentially ignore and the next word defines the field type
VARCHAR	Emit next field as VARCHAR instead of CHAR
VAR-LENGTH	Emit next field as VARCHAR instead of CHAR
WHEN fld op val	fld is some previous field in the record, val is a numeric value of alpha value in quotes, op is one of =, !=, >, >=, <, <=. val may be other to indicate other cases. If the condition is true then the following field/group format is used.
AND fld op val	A continuation of the previous WHEN logically connect as an AND condition
OR fld op val	A continuation of the previous WHEN logically connect as an OR condition

Name	**GnuCOBOL Compile time Option Descriptions**
-fsqldb=name	name indicates which database is to be used. Choose one of ODBC, MySQL, MSSQL, OCI, Oracle10, Oracle11, Oracle12.

-fsqlschema=name	name is sub-directory into which the filename.ddl holding CREATE TABLE is written as well as the filename.xd holding the data mapping information. If omitted then the ddl/xd files are written to the base schema_dir directory.

```
Environment
Variable          runtime.cfg  GnuCOBOL Run time Option Descriptions

COB_SCHEMA_DIR    schema_dir   Define location where the filename.xd files are
                               stored. Default: $COB_CONFIG_DIR/schema
COB_SCHEMA_NAME   schema_name  Database schema name
COB_SCHEMA_UID    schema_uid   Database User-id for connecting to the database
COB_SCHEMA_PWD    schema_pwd   Password for database User-Id
COB_SCHEMA_DSN    schema_dsn   DSN name for database
COB_DEBUG_LOG                  Parameters for GnuCOBOL logging feature.
                               Module type is db
```

Implementation Plan - GnuCOBOL Developer Notes

To support $XFD directives changes were made to pplex.l, ppparse.y and scanner.l. Also codegen.c and field.c will need changes.

Additional flags may have to be added to cb_field to retain the XFD directive information.

In codegen.c (output_file_initialization) depending on use of XFD and compile options is the location to emit the CREATE TABLE and the data description file. The cob_file will get some new flags for io_routine to indicate the file is really handled by ODBC/OCI.

A new fileio routine (fodbc.c and later foci.c) were developed following the recently developed interface for multiple I/O handlers. The cob_load_xfd routine loads the data description file into internal structures. Multiple schemas could have the same table name/structure for development, QA, production, etc. This could be indicated via runtime.cfg options and/or environment variables. IO_asgname may have format=odbc or format=oci to indicate which method is used to access the SQL table for the file. You may also specify table=sql_table_name to override the default SQL table.

At run-time when the file is opened the file description is loaded into memory for fast access and conversion of the data between COBOL data type and SQL column data. Since SQL tables are usually collected under a database schema well need a compile option to generate the external file description under either a default directory or under a given schema name. At run-time an environment variable and/or runtime.cfg option could be used to define which database schema is to be used along with the database user-id and password. Often a database user-id has a default schema associated with it.

The module fsqlxfd.c contains routines common to both fodbc.c and foci.c such as loading of the filename.xd information and conversion of the data between SQL and COBOL data types. When the file is OPENed and the table is found to not exist, then cob_load_ddl loads the filename.ddl into memory for the OCI/ODBC routine to submit the CREATE TABLE/INDEX statements as required. As the table is loaded, if it contains column types

not supported by the current database then there is a translation done during the load process.

Format of filename.xd

GnuCOBOL has its own format instead of using what ACUCOBOL had. The first character on each line indicates what the line is. Lines starting with # or * are comments. Each field is separated by a comma.

Name	**Header Line Description**
H	Is a header
1	Version number
table	SQL table name
,	For character used as decimal, default ,
.	For character used as comma, default .
Hex	Indicate how numeric signs are handled
num	File type, 3 for INDEXED, 2 for RELATIVE

Name	**Label Description**
L	Is a Label
Num	Numeric label, may be referenced by goto or condition

Name	**Goto Description**
G	Is a Goto
Num	Numeric label to transfer to

Name	**Condition Description**
C	Is a Condition to be tested (Appear in postfix sequence) Complex WHEN clauses become multiple Condition tests
Label	If condition test is true, then goto this label, 0 indicates more coming, non-zero indicates end of the WHEN condition to be tested
Opcode	>=, >, <=, <, =, !=, &&, ||, !
Operand	Usually the data field
Value	String in quotes or numeric value

Name	**Date Format Description**
D	Date format definition
num	Unique number for this format
date	Date format string
num	Total number of digits in this date field
num	1 if DATE present else 0
num	1 if TIME present else 0
yy rule	+ means add yyAdj to YY, % defines pivot point
num	Value for adjusting the Year
P:L	Year Position and Length within field
P:L	Month Position and Length within field

P:L	Day Position and Length within field
P:L	Hours Position and Length within field
P:L	Minutes Position and Length within field
P:L	Seconds Position and Length within field
P:L	Century date Position and Length within field

You may define how your application stores DATE information. The database will always expect dates to be in a full YYYYMMDD format and date/time to be in YYYYMMDDHHMISS format. Date fields could be defined like the following:

```
$XFD DATE "YYYYMMDD"
$XFD DATE "YY%60MMDD"
$XFD DATE "YYY+1800MMDD"
$XFD DATE "YYMMDDCC"
```

The Y is a place holder for YEAR, MM for month, DD for day, HH hour, MI minutes, SS seconds, CC for century. If the Ys are followed by % then the digits after the % defines a pivot year used to map the YY value into a 4 digit year. In the above example if the YY value is below 60 then it is 19YY else 20YY

If the Y's are followed by + then the digits after the + are added to the Y value. In above example, the year is 1800 + YYY value. There is a limit of 16 different DATE formats per record.

If the day is defined like DDD (3 Ds) then it is taken to be the day of the year. For example:

```
$XFD DATE "YYYYDDD"
```

Name	**Data Description**
F	Define data Field
num	Offset from record to start of COBOL data field
num	Bytes occupied by COBOL data field
num	Type of data field
	1 Arbitrary Binary Data
	2 PIC S9 COMP-5
	3 PIC 9 COMP-5
	4 PIC 9 COMP-6
	5 PIC S9 BINARY/COMP/COMP-4
	6 PIC 9 BINARY/COMP/COMP-4
	7 PIC x COMP-X
	8 COMP-1/COMP-2
	9 PIC S9 COMP-3/PACKED DECIMAL
	10 PIC 9 COMP-3/PACKED DECIMAL
	11 PIC S9 SIGN LEADING
	12 PIC S9 SIGN LEADING SEPARATE

13 PIC S9
14 PIC 9 SIGN TRAILING
15 PIC 9 SIGN TRAILING SEPARATE
16 PIC 9
17 PIC A
18 PIC X National characters
19 PIC X Wide characters
20 PIC X
21 PIC X - VARCHAR

num	Bytes needed to store as SQL data
num	Digits in numeric field
num	Scale (or decimal places)
num	Unique id of date format (see D)
num	COBOL level number
name	SQL Column name used for data field

Name	**Key Definitions Description**
K	Define an index
num	Index number (1 16)
dup	Y if duplicates allowed, else N
sup	Y if key may be suppressed, else N
char	Suppress character either as c or 0xHH or string
columns	Comma separate list of column names in order that make up the index

Example Records to Table With the following:

```
           SELECT OPTIONAL TSPFILE
           ASSIGN TO "testsql"
           ORGANIZATION INDEXED ACCESS DYNAMIC
           RECORD KEY IS PRIME-KEY
           SOURCE IS CM-CUST-NUM

           ALTERNATE RECORD KEY IS SPLIT-KEY2
           SOURCE IS CM-TELEPHONE,CM-MACHINE WITH DUPLICATES
                SUPPRESS WHEN "900"

           ALTERNATE RECORD KEY IS SPLIT-KEY3
           SOURCE IS CM-DISK,CM-DP-MGR,CM-MACHINE WITH DUPLICATES
                SUPPRESS WHEN ALL "*"
           FILE STATUS IS CUST-STAT
           .

           SELECT FLATFILE ASSIGN "relfile"
           ORGANIZATION RELATIVE
```

```
           ACCESS IS RANDOM RELATIVE KEY IS REC-NUM
           FILE STATUS IS CUST-STAT.

           SELECT FLATSEQ ASSIGN "relfile"
           ORGANIZATION RELATIVE
           ACCESS IS SEQUENTIAL RELATIVE KEY IS REC-NUM
           FILE STATUS IS CUST-STAT.
```

Note that FLATFILE and FLATSEQ are the same file called relfile with a different ACCESS. Given the following record description:

```
      $XFD ALL
       FD  FLATFILE
           BLOCK CONTAINS 5 RECORDS.

       01  FLAT-RECORD.
           10  RL-CUST-NUM                    PICTURE X(8).
           10  RL-COMPANY                     PICTURE X(25).
           10  RL-TRAILER                     PICTURE X(16).

       FD  FLATSEQ
           BLOCK CONTAINS 5 RECORDS.

       01  RS-RECORD.
           10  RS-CUST-NUM                    PICTURE X(8).
           10  RS-COMPANY                     PICTURE X(25).
           10  RS-TRAILER                     PICTURE X(16).

       FD  TSPFILE
           BLOCK CONTAINS 5 RECORDS.

      $XFD NAME=tspfilex
       01  TSPFL-RECORD.
           05  TSPFL-REC.
      $XFD USE GROUP CUSTNUM
           10  CM-CUST-NUM.
             15  CM-CUST-PRE                  PICTURE X(3).
             15  CM-CUST-NNN                  PICTURE X(5).
           10  CM-STATUS                      PICTURE X.
           10  CM-COMPANY                     PICTURE X(25).
      $XFD USE GROUP VAR_LENGTH custaddr
           10  CM-ADDRESS.
             15  CM-ADDRESS-1                 PICTURE X(25).
             15  CM-ADDRESS-2                 PICTURE X(25).
```

```
           15  CM-ADDRESS-3                  PICTURE X(25).
         10  CM-TELEPHONE                    PICTURE 9(10).
         10  CM-DP-MGR                       PICTURE X(25).
         10  CM-MACHINE                      PICTURE X(8).
         10  CM-MEMORY                       PICTURE X(4).
    $XFD WHEN (CM-STATUS = 'A' && CM-TELEPHONE > 100)
    $XFD AND CM-MACHINE = 'B' || CM-COMPANY = ' '
         10  CM-MEMORYX REDEFINES CM-MEMORY.
           15  CM-MEMSZ                      PICTURE 9(2).
           15  CM-MEMUNIT                    PICTURE X(2).
         10  CM-DISK                         PICTURE X(8).
         10  CM-TAPE                         PICTURE X(8).
    $XFD WHEN CM-STATUS = 'X'
         10  CM-TAPEX REDEFINES CM-TAPE      PICTURE 9(8).
    $XFD WHEN CM-STATUS = 'Y'
         10  CM-TAPEY REDEFINES CM-TAPE      PICTURE 9(6)V99.
         10  CM-NO-TERMINALS                 PICTURE 9(5) BINARY.
         10  CM-COMPX                        PICTURE XXX COMP-X.
         10  CM-COMP5                        PICTURE 9(7) COMP-5.
         10  CM-COMP1                        COMP-1.
         10  CM-COMP2                        COMP-2.
         10  CM-PRICE                        PICTURE 9(3)V99 COMP-3.
         10  CM-PRICES                       PICTURE S9(5)V99.
    $XFD DATE "MMDDYYYY"
         10  CM-DATE                        PICTURE 9(8) COMP-3.
    $XFD DATE "YYMMDDCC"
         10  CM-DATE2                        PICTURE 9(8) COMP-3.
```

And the resulting SQL table(s) would look like:

tspfilex.ddl --

```
CREATE TABLE tspfilex (
   custnum                                 CHAR(8) NOT NULL,
   status                                  CHAR(1),
   company                                 CHAR(25),
   custaddr                                VARCHAR(75),
   telephone                               DECIMAL(10) NOT NULL,
   dp_mgr                                  CHAR(25) NOT NULL,
   machine                                 CHAR(8) NOT NULL,
   memory                                  CHAR(4),
   memsz                                   DECIMAL(2),
   memunit                                 CHAR(2),
   disk                                    CHAR(8) NOT NULL,
```

```
   tape                                    CHAR(8),
   tapex                                   DECIMAL(8),
   tapey                                   DECIMAL(8,2),
   no_terminals                            DECIMAL(5),
   compx                                   DECIMAL(8),
   comp5                                   DECIMAL(7),
   comp1                                   FLOAT(23),
   comp2                                   FLOAT(53),
   price                                   DECIMAL(5,2),
   prices                                  DECIMAL(7,2),
   date_x                                  DATE,
   date2                                   DATE
);
CREATE UNIQUE INDEX pk_tspfilex ON tspfilex (custnum);
CREATE INDEX k1_tspfilex ON tspfilex (telephone,machine);
CREATE INDEX k2_tspfilex ON tspfilex (disk,dp_mgr,machine);

relfile.ddl --

CREATE TABLE relfile (
   cust_num                                CHAR(8),
   company                                 CHAR(25),
   trailer                                 CHAR(16),
   rid_relfile                             BIGINT PRIMARY KEY
);



The tspfilex.xd description for this follows:

H,1,tspfilex,2,',',',',0,3
D,1,'MMDDYYYY',8,1,0,,0,4:4,0:2,2:2,0:0,0:0,0:0,0:0
D,2,'YYMMDDCC',8,1,0,,0,0:2,2:2,4:2,0:0,0:0,0:0,6:2
F,0000,0008,20,0009,0,0,,10,custnum
F,0008,0001,20,0002,0,0,,10,status
F,0009,0025,20,0026,0,0,,10,company
F,0034,0075,21,0076,0,0,,10,custaddr
F,0109,0010,16,0013,10,0,,10,telephone
F,0119,0025,20,0026,0,0,,10,dp_mgr
F,0144,0008,20,0009,0,0,,10,machine
C,0,=,status,'A'
C,0,>,telephone,100
C,0,&&
C,0,=,machine,'B'
C,0,=,company,' '
C,0,||
```

```
C,2,&&
F,0152,0004,20,0005,0,0,,10,memory
G,3
L,2
F,0152,0002,16,0005,2,0,,15,memsz
F,0154,0002,20,0003,0,0,,15,memunit
L,3
F,0156,0008,20,0009,0,0,,10,disk
C,5,=,status,'X'
C,6,=,status,'Y'
F,0164,0008,20,0009,0,0,,10,tape
G,7
L,5
F,0164,0008,16,0011,8,0,,10,tapex
G,7
L,6
F,0164,0008,16,0011,8,2,,10,tapey
L,7
F,0172,0004,06,0013,5,0,,10,no_terminals
F,0176,0003,07,0011,8,0,,10,compx
F,0179,0004,03,0013,7,0,,10,comp5
F,0183,0004,08,0036,15,8,,10,comp1
F,0187,0008,08,0036,34,17,,10,comp2
F,0195,0003,10,0008,5,2,,10,price
F,0198,0007,13,0010,7,2,,10,prices
F,0205,0005,10,0032,8,0,1,10,date_x
F,0210,0005,10,0032,8,0,2,10,date2
K,0,N,N,,custnum
K,1,Y,Y,"900",telephone,machine
K,2,Y,Y,0x2A,disk,dp_mgr,machine


The relfile.xd description for this follows:

H,1,relfile,0,',',.',0,2
F,0000,0008,20,0009,0,0,,10,cust_num
F,0008,0025,20,0026,0,0,,10,company
F,0033,0016,20,0017,0,0,,10,trailer
F,0049,0004,03,0015,12,0,,00,rid_relfile
K,0,N,N,,rid_relfile
```

Application Schema

Sometimes COBOL programs will have local views of the data by copying a record to some other group item which redefines the data differently. In some cases these hidden redefines will cause problems for SQL columns. For example, if a field is defined as PIC 9(x) but sometimes the same location in the record contains non-numeric data, then SQL

will reject the storing of such data as it violates the strict data typing. By using the $XFD WHEN you are able to define which a portion of the record has one view other another. The I/O logic will then only copy the valid column of data.

It may therefore be a good idea to collect all of the file definitions into a single module, verify that all possible redefines are identified. The $XFD ALL directives indicates that such a module is being compiled. Think of this module as defining all of the master data files in an application which will be migrated into SQL tables.

Check the DDL

It is a good idea to check the DDL generated by the GnuCOBOL compiler and make sure it suits your local database guidelines. You may also need to add local directives such as a STORAGE clause for Oracle. You should then use the appropriate tool (eg. Mysql or sqlplus) and manually create all of the tables to be used by an application. There may also be considerations for what database user-id has permission to create tables, read/write tables etc.

Automatic Table Creation

At OPEN time, the filename.xd is processed and if not present that is considered a serious error (30) is returned.

At OPEN time, a check is done to see if the table exists and if not, the run-time code then reads the filename.ddl and submits it to the database to create the table and indexes. If the filename.ddl is not present then OPEN is given an error (30). If runtime.cfg option create_table is true then the runtime will attempt to recreate filename.ddl from the information in filename.xd.

Taken from Development Notes by Ron Norman February 2020.

End of Chapter 13 — Programming for XFD

Appendix A - Glossary of Terms

Alphabetic Data Item
: A data item whose `PICTURE` clause allows it to contain only upper- and/or lower-case letters. See [PICTURE], page 194.

Alphanumeric Data Item
: A data item whose `PICTURE` clause allows it to contain absolutely any character whatsoever. See [PICTURE], page 194. Group items (see [Structured Data], page 14) are also implicitly considered to be alphanumeric data items.

Alphanumeric Literal
: A string of characters enclosed within a pair of quotation marks (‘"’) or apostrophes (‘’’). See [Alphanumeric Literals], page 35.

Called Program
: Another way to refer to a subprogram. Note that a called program may also be a calling program.

Calling Program
: A program that executes a subprogram. Note that a calling program may also be a called program.

Collating Sequence
: The sequence in which the characters that are acceptable to a computer are ordered for purposes of all types of sorting, merging, comparing, and processing. GnuCOBOL programs may utilize standard character-set collating sequences (such as that defined by the ASCII or EBCDIC character sets) or programmer-defined custom sequences as specified in the OBJECT-COMPUTER paragraph (section 4.1.2) and defined in the SPECIAL-NAMES paragraph (section 4.1.4).

Compilation Group
: The collection of all compilation units being compiled by a single execution of the GnuCOBOL compiler.

Compilation Unit
: A single source file being compiled by the GnuCOBOL compiler. A compilation unit may contain one or more programs.

Control Break
: An event that is triggered when a control field on an RWCS-generated report changes value. It is these events that trigger the generation of control heading and control footing groups.

Control Field
: A field of data being presented within a detail group; as the various detail groups that comprise the report are presented, they are presumed to appear in sorted sequence of the control fields contained within them. As an example, a department-by-department sales report for a chain of stores would probably be sorted by store number and – within like store numbers – be further sorted by department number. The store number will undoubtedly serve as a control

field for the report, allowing control heading groups to be presented before each sequence of detail groups for the same store and control footing groups to be presented after each such sequence.

Control Footing
: A report group that appears immediately after one or more detail groups of an RWCS-generated report. Such are produced automatically as a result of a control break. This type of group typically serves as a summary of the detail group(s) that precede it, as might be the case on a sales report for a chain of stores, where the detail groups documenting sales for each department (one department per detail group) from the same store might be followed by a control footing that provides a summation of the department-by-department sales for that store.

Control Heading
: A report group that appears immediately before one or more detail groups of an RWCS-generated report. Such are produced automatically as a result of a control break. This type of group typically serves as an introduction to the detail group(s) that follow, as might be the case on a sales report for a chain of stores, where the detail groups documenting sales for each department (one department per detail group) from the same store might be preceded by a control heading that states the full name and location of the store.

Control Hierarchy
: The natural hierarchy of control breaks within a RWCS-controlled report based upon the manner in which the data the report is being generated from is sorted.

Copybook
: A segment of program code that may be utilized by multiple programs simply by having that program use the `COPY` statement to import that code into the program. Although similar to the "include" facility present in many other programming languages, the COBOL copybook mechanism is actually considerably more powerful. See [Copybooks], page 14, for a general discussion. See [COPY], page 63, for the specifics of the `COPY` statement.

Data Item
: A contiguous area of storage within the memory space of a program that may be referenced, by name, in a COBOL program. Other programming languages use the term variable, property or attribute to describe the same concept. See [Structured Data], page 14.

Detail Group
: A report group that contains the detailed data being presented for the report.

Detail Report
: An RWCS-generated report to which at least one type of detail group is presented.

Division
: A collection of zero, one, or more sections of paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules.

Each division consists of the division header and the related division body. There are four divisions in a GnuCOBOL program: Identification, Environment, Data, and Procedure (coded in that sequence). See [Program Structure], page 31.

Dynamic Subprogram
: A subprogram whose executable object code is contained in a different executable file as its calling program. Dynamic subprograms are therefore loaded into memory as needed.

Elementary Item
: A data item that isn't itself comprised of other data items. See [Structured Data], page 14.

Entry-point
: A spot in the procedure division where a program may begin execution when it is executed from the operating system, invoked as a user-defined function or called by another program.

  Every program has at least one entry-point — known as the primary entry-point — which corresponds to the first executable statement in the procedure division following the declaratives area, if any.

  Additional entry-points may be defined via the `ENTRY` statement (see [ENTRY], page 318).

Entry-point Name
: Every entry-point has a name. That name must be unique for all programs that comprise an executable program. Entry-point names are defined using a subroutine's `PROGRAM-ID` paragraph, a user-defined function's `FUNCTION-ID` paragraph or via `ENTRY` (see [ENTRY], page 318) statements coded in a subprogram's procedure division.

Executable File
: The GnuCOBOL compiler can create operating-system appropriate files that may be executed directly from the operating system environment. On Windows systems, these will be `.exe` files whereas on UNIX systems they will have no specific extensions. The compiler's `-x` switch is used to create executable files. Only main programs should be compiled in this manner.

Execution Thread
: The complete set of executable code that is run during the execution of a program. This includes the main program as well as all executed subprograms, including those that are both dynamically and statically loaded.

Figurative Constants
: GnuCOBOL, like other COBOL implementations, supports a number of reserved words that may be used to represent a specific literal value. These are known as figurative constants. See [Figurative Constants], page 36, for more information.

Fixed Format Mode
: A mode of the GnuCOBOL compiler's operation where source statements are constrained to meeting the pre-2002 standard of limiting COBOL statements to 80 columns, with various columns having limitations as to what sort of COBOL syntax could be specified in them. See [Format of Program Source Lines], page 28, for more information.

Free Format Mode
: A mode of the GnuCOBOL compiler's operation where source statements are allowed to be as long as 255 characters, with no restrictions or requirements as to in which columns various syntax elements must appear. See [Format of Program Source Lines], page 28, for more information.

Group Item
: A hierarchical data structure where the group item — itself a data item — actually consists of two or more other contiguously allocated data items. For example, `Employee-Name` could be a 35-character data item consisting of a 20-character `Last-Name` data item followed by a 14-character `First-Name` and a 1-character `Middle-Initial`. See [Structured Data], page 14.

Hexadecimal Alphanumeric Literal
: These are alphanumeric literals whose character sequence is specified by hexadecimal value. These literals are formed by a quote- or apostrophe-delimited sequence of an even number of hexadecimal digits (upper- or lower-case), prefixed with the letter '`X`' (also upper- or lower-case). For example, the character string "`Demo`" could be specified as the hexadecimal alphanumeric literal `X'44656D6F'`, assuming the ASCII character set. See [Alphanumeric Literals], page 35.

Hexadecimal Numeric Literal
: A numeric literal whose value is specified by hexadecimal value. These literals are formed by a quote- or apostrophe-delimited sequence of from 1 to 16 hexadecimal digits (upper- or lower-case), prefixed with the letter '`H`' (also upper- or lower-case). For example, the number 123456 could be specified as the hexadecimal numeric literal `H'01E240'`. See [Numeric Literals], page 34.

Identifiers
: These are data items a COBOL program will be working with. The vast majority of identifiers are defined by the user (programmer) while a few are predefined by the GnuCOBOL compiler. Identifiers pre-defined by the compiler are referred to as special registers. Other programming languages generally refer to identifiers as "variables".

Imperative Statement
: Either a statement that begins with a non decision-making verb and specifies an unconditional action to be taken or a conditional verb such as `IF` or `EVALUATE`, delimited by its explicit scope terminator (such as `END-IF` or `END-EVALUATE`). An imperative statement can consist of a sequence of imperative statements.

Intrinsic Function

A built-in routine that accepts arguments and returns a value; syntactically, these may be used most places where GnuCOBOL identifiers are valid. See [Intrinsic Functions], page 421, for documentation on all the GnuCOBOL intrinsic functions.

Level Number

A 1- or 2-digit number that indicates the hierarchical position of a data item in a group item or the special properties of a data description entry.

Level numbers in the range 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record. Level numbers in the range 1 through 9 can be written either as a single digit or as a zero followed by the significant digit.

Level numbers 66, 77, 78 and 88 identify special properties of a data description entry.

Literal A generic term used for a constant value coded in a program that may be either a numeric literal or an alphanumeric literal.

Main program

A program that is executed directly from an operating system or shell event. Main programs are not executed from other programs (i.e. they are not called programs).

National Character set

A character set that supports symbols using other than the traditional Roman alphabet symbols used by the ASCII character set. Typically, such a character set uses a UTF-16 (i.e. 16 bits-per-character) encoding of the Unicode character set.

Support for national character sets in GnuCOBOL is currently only partially implemented, and the compile- and run-time effect of using the `N` symbol in a `PICTURE` (see [PICTURE], page 194) clause to define a field as containing national characters is the same as if `X(2)` had been coded, with the additional effect that such a field will qualify as a `NATIONAL` or `NATIONAL-EDITED` field on an `INITIALIZE` (see [INITIALIZE], page 339) statement.

Numeric Data Item

A data item whose `PICTURE` clause allows it to contain only the numeric digit characters `0-9` (signed or unsigned), or a data item whose `PICTURE`/`USAGE` combination allow it to contain actual binary numbers in integer, fixed-point, floating-point or packed-decimal format. Numeric data items are the only ones that may be used as table subscripts or as source arguments on arithmetic statements. `PICTURE` (see [PICTURE], page 194), or `USAGE` (see [USAGE], page 232).

Numeric Edited Data Item

An otherwise numeric data item whose `PICTURE` (see [PICTURE], page 194) clause also contains any of the editing symbols '`$`', '`*`', '`+`', '`,`', '`-`', '`.`', '`/`', '`0`'

(zero), 'B', 'CR', 'DB' or 'Z'. Numeric edited data items are not eligible to serve as table subscripts or source arguments on arithmetic statements.

Numeric Literal
: A numeric constant. See [Numeric Literals], page 34.

Page Footing
: A report group that appears at the bottom of every page of an RWCS-generated report. Information typically found within such a report group might be:
  - The date the report was generated
  - The current page number of the report

Page Heading
: A report group that appears at the top of every page of an RWCS-generated report. Information typically found within such a report group might be:
  - A title for the report
  - The date the report was generated
  - The current page number of the report
  - Column headings describing the fields within the detail group(s)

Primary Entry-Point
: See entry-point.

Procedure
: All executable code statements within a single procedure division paragraph or section.

Procedure name
: A programmer-defined section or paragraph name in the procedure division assigned to a procedure. Procedure names serve as a means by which a statement may refer to the statements that follow the procedure name.

Program
: A GnuCOBOL main program or subprogram.

Qualification
: The process of establishing a unique reference to a data item whose name is duplicated in a program. This takes the form of using the duplicated data name and the name of any of its parent data items, connected by `OF` or `IN` such that the combination of those two data names is unique within the program.

Record
: A group item that is not part of a higher-level group item. See [Data Definition Principles], page 120. An elementary item with a level number of 01 can also be referred to as a record if its definition occurs in the file section, provided that its definition does not include the `CONSTANT` attribute. See [FILE-SECTION-Data-Item], page 127.

Report Footing
: A report group that occurs only once in an RWCS-generated report — as the very last presented report group of the report. These typically serve as a visual indication that the report is finished.

Report Group
: One or more consecutive lines on a report that serve a common informational purpose or function. For example, lines of text that are displayed at the top or bottom of every printed page of a report.

Report Heading
: A report group that occurs only once in an RWCS-generated report — as the very first presented report group of the report. These typically serve as an introduction to the report.

Reserved Word
: A word coded in a GnuCOBOL program without any quote or apostrophe characters around it (which would have transformed that sequence of characters into a literal string) which has a very specific meaning to the compiler. See [Language Reserved Words], page 9, for a general discussion of the concept. Appendix C for a complete list of GnuCOBOL reserved words.

Sentence
: An arbitrarily long sequence of statements terminated by a period.

Special Registers
: Special data items that are automatically defined for your use by the GnuCOBOL compiler. See [Special Registers], page 265, for a complete list.

Statement
: A single executable COBOL instruction. All statements start with a verb (`DISPLAY`, `IF`, `MOVE`, ...) which is followed by the operands and additional syntax elements that describe the actions to be performed.

Static Subprogram
: A subprogram whose executable object code is part of the same executable file as its calling program. Static subprograms are therefore loaded into memory at the same time as their caller.

Subprogram
: A program invoked directly by another program in such a manner that it may return control back to the other program, directly back to the point where the subprogram was invoked.

Subroutine
: A subprogram executed from another via a GnuCOBOL `CALL` (see [CALL], page 293) statement (or the equivalent in whatever programming language that other program was written in).

Summary Report
: An RWCS-generated report to which no detail groups are presented.

User-Defined Function
: A subprogram written in GnuCOBOL that is executed in a syntactically-similar manner to that by which the various built-in intrinsic functions are executed.

User-Defined Names
: Either the name of an identifier or a procedure in the program. GnuCOBOL limits user-defined names to a maximum of 31 characters taken from the set

of numeric digits, upper- and lower-case letters, hyphens and underscores. A user-defined name may neither begin nor end with a hyphen or underscore. User-defined names used as file names may additionally not begin with a digit although - unlike many other programming languages - user-defined names used as identifiers or procedure names may.

Verb The first reserved word of a COBOL statement.

Zero-Delimited Alphanumeric Literals
An alphanumeric literal prefixed with an upper- or lower-case 'Z' character — for example, `Z'ABC'`. These literals are one character longer than the value within apostrophes or quotes would make them appear. The extra character (the last character) will be a null character (comprised entirely of zero bits). These literals are ideal when defining or assigning values to alphanumeric data items that will be passed as arguments to a C subroutine. See [Alphanumeric Literals], page 35.

End of Appendix A — Glossary of Terms

Appendix B - Reserved Word Lists

See Appendix C for the complete lists of ALL reserved words of all types.

End of Appendix B — Reserved Word Lists

Appendix C1 - Grouped Word Lists by feature and function

The following is the complete list of ALL reserved words in the 25 January 2023 at 18:20 GMT. build of GnuCOBOL 3.2. Even though the functionality behind some of these words may not be implemented in this version of GnuCOBOL, none may be used as any user-defined name. This list includes ALL reserved, intrinsics, mnemonics and system and shows some 1100+ words in total. In addition there are the arithmetic and relational symbols see 1.3.15 as well as extra words that can be added and existing words removed by the use of the –std file content see the specific one used for each sub-set.

The following list of reserved words was extracted from `cobc --list-reserved` and shows the reserved words, an implementation

Please notice: This list is highly specific to the option `-std=`*`dialect`* and reserved word options (`-freserved=`*`word`*, `-fno-reserved=`*`word`*) in effect. You can get the list for a given *dialect* by calling `cobc -std=`*`dialect`* `--list-reserved`.

Common reserved words

Reserved word	Implemented	Aliases
3-D	Yes (C/S)	
ABSENT	Yes	
ACCEPT	Yes	
ACCESS	Yes	
ACTION	Yes (C/S)	
ACTIVATING	No (C/S)	
ACTIVE-CLASS	No	
ACTIVE-X	Yes (C/S)	
ACTUAL	Yes (C/S)	
ADD	Yes	
ADDRESS	Yes	
ADJUSTABLE-COLUMNS	Yes (C/S)	
ADVANCING	Yes	
AFTER	Yes	
ALIGNED	No	
ALIGNMENT	Yes (C/S)	
ALL	Yes	
ALLOCATE	Yes	
ALLOWING	Yes (C/S)	
ALPHABET	Yes	
ALPHABETIC	Yes	
ALPHABETIC-LOWER	Yes	
ALPHABETIC-UPPER	Yes	
ALPHANUMERIC	Yes	
ALPHANUMERIC-EDITED	Yes	
ALSO	Yes	
ALTER	Yes	

ALTERNATE	Yes	
AND	Yes	
ANUM	No	(C/S)
ANY	Yes	
ANYCASE	No	
APPLY	Yes (C/S)	
ARE	Yes	
AREA	Yes	AREAS
AREAS	Yes	AREA
ARGUMENT-NUMBER	Yes	
ARGUMENT-VALUE	Yes	
ARITHMETIC	Yes (C/S)	
AS	Yes	
ASCENDING	Yes	
ASCII	Yes (C/S)	
ASSIGN	Yes	
AT	Yes	
ATTRIBUTE	Yes (C/S)	
ATTRIBUTES	Yes (C/S)	
AUTHOR	Yes (C/S)	
AUTO	Yes (C/S)	AUTO-SKIP, AUTOTERMINATE
AUTO-DECIMAL	Yes (C/S)	
AUTO-SKIP	Yes	AUTO, AUTOTERMINATE
AUTO-SPIN	Yes (C/S)	
AUTOMATIC	Yes	
AUTOTERMINATE	Yes	AUTO, AUTO-SKIP
AWAY-FROM-ZERO	Yes (C/S)	
B-AND	No	
B-NOT	No	
B-OR	No	
B-SHIFT-L	Yes	
B-SHIFT-LC	Yes	
B-SHIFT-R	Yes	
B-SHIFT-RC	Yes	
B-XOR	No	
BACKGROUND-COLOR	Yes (C/S)	BACKGROUND-COLOUR
BACKGROUND-COLOUR	Yes	BACKGROUND-COLOR
BACKGROUND-HIGH	Yes	
BACKGROUND-LOW	Yes	
BACKGROUND-STANDARD	Yes	
BACKWARD	Yes (C/S)	
BAR	Yes (C/S)	
BASED	Yes	
BEEP	Yes	BELL
BEFORE	Yes	
BELL	Yes (C/S)	BEEP

BINARY	Yes	
BINARY-C-LONG	Yes	
BINARY-CHAR	Yes	
BINARY-DOUBLE	Yes	BINARY-LONG-LONG
BINARY-INT	Yes	BINARY-LONG
BINARY-LONG	Yes	BINARY-INT
BINARY-LONG-LONG	Yes	BINARY-DOUBLE
BINARY-SEQUENTIAL	Yes (C/S)	
BINARY-SHORT	Yes	
BIT	Yes	
BITMAP	Yes (C/S)	
BITMAP-END	Yes (C/S)	
BITMAP-HANDLE	Yes (C/S)	
BITMAP-NUMBER	Yes (C/S)	
BITMAP-START	Yes (C/S)	
BITMAP-TIMER	Yes (C/S)	
BITMAP-TRAILING	Yes (C/S)	
BITMAP-TRANSPARENT-COLOR	Yes (C/S)	
BITMAP-WIDTH	Yes (C/S)	
BLANK	Yes	
BLINK	Yes (C/S)	
BLOCK	Yes	
BOOLEAN	No	
BOTTOM	Yes	
BOX	Yes (C/S)	
BOXED	Yes (C/S)	
BULK-ADDITION	Yes (C/S)	
BUSY	Yes (C/S)	
BUTTONS	Yes (C/S)	
BY	Yes	
BYTE-LENGTH	Yes (C/S)	
C	Yes (C/S)	
CALENDAR-FONT	Yes (C/S)	
CALL	Yes	
CANCEL	Yes	
CANCEL-BUTTON	Yes (C/S)	
CAPACITY	Yes (C/S)	
CARD-PUNCH	Yes (C/S)	
CARD-READER	Yes (C/S)	
CASSETTE	Yes (C/S)	
CCOL	Yes (C/S)	
CD	Yes	
CELL	Yes (C/S)	CELLS
CELL-COLOR	Yes (C/S)	
CELL-DATA	Yes (C/S)	
CELL-FONT	Yes (C/S)	

CELL-PROTECTION	Yes (C/S)	
CELLS	Yes	CELL
CENTER	Yes (C/S)	
CENTERED	Yes (C/S)	
CENTERED-HEADINGS	Yes (C/S)	
CENTURY-DATE	Yes (C/S)	
CF	Yes	
CH	Yes	
CHAIN	No	
CHAINING	Yes	
CHANGED	Yes (C/S)	
CHARACTER	Yes	
CHARACTERS	Yes	
CHECK-BOX	Yes (C/S)	
CLASS	Yes	
CLASS-ID	No	
CLASSIFICATION	Yes (C/S)	
CLEAR-SELECTION	Yes (C/S)	
CLINE	Yes (C/S)	
CLINES	Yes (C/S)	
CLOSE	Yes	
COBOL	Yes (C/S)	
CODE	Yes	
CODE-SET	Yes	
COL	Yes	
COLLATING	Yes	
COLOR	Yes	
COLORS	Yes (C/S)	COLOURS
COLOURS	Yes	COLORS
COLS	Yes	
COLUMN	Yes	
COLUMN-COLOR	Yes (C/S)	
COLUMN-DIVIDERS	Yes (C/S)	
COLUMN-FONT	Yes (C/S)	
COLUMN-HEADINGS	Yes (C/S)	
COLUMN-PROTECTION	Yes (C/S)	
COLUMNS	Yes	
COMBO-BOX	Yes (C/S)	
COMMA	Yes	
COMMAND-LINE	Yes	
COMMIT	Yes	
COMMON	Yes	
COMMUNICATION	Yes	
COMP	Yes	COMPUTATIONAL
COMP-0	Yes	COMPUTATIONAL-0
COMP-1	Yes	COMPUTATIONAL-1

COMP-10	Yes	COMP-10, DOUBLE, FLOAT-LONG
COMP-15	Yes	COMP-15, DOUBLE, FLOAT-LONG
COMP-2	Yes	COMPUTATIONAL-2
COMP-3	Yes	COMPUTATIONAL-3
COMP-4	Yes	COMPUTATIONAL-4
COMP-5	Yes	COMPUTATIONAL-5
COMP-6	Yes	COMPUTATIONAL-6
COMP-9	Yes	FLOAT, FLOAT-SHORT
COMP-N	Yes	COMPUTATIONAL-N
COMP-X	Yes	COMPUTATIONAL-X
COMPUTATIONAL	Yes	COMP
COMPUTATIONAL-0	Yes	COMP-0
COMPUTATIONAL-1	Yes	COMP-1
COMPUTATIONAL-2	Yes	COMP-2
COMPUTATIONAL-3	Yes	COMP-3
COMPUTATIONAL-4	Yes	COMP-4
COMPUTATIONAL-5	Yes	COMP-5
COMPUTATIONAL-6	Yes	COMP-6
COMPUTATIONAL-N	Yes	COMP-N
COMPUTATIONAL-X	Yes	COMP-X
COMPUTE	Yes	
CONDITION	Yes	
CONFIGURATION	Yes	
CONSTANT	Yes	
CONTAINS	Yes	
CONTENT	Yes	
CONTINUE	Yes	
CONTROL	Yes	
CONTROLS	Yes	
CONVERSION	Yes (C/S)	
CONVERTING	Yes	
COPY	Yes	
COPY-SELECTION	Yes (C/S)	
CORE-INDEX	Yes (C/S)	
CORR	Yes	CORRESPONDING
CORRESPONDING	Yes	CORR
COUNT	Yes	
CRT	Yes	
CRT-UNDER	Yes	
CSIZE	Yes (C/S)	
CURRENCY	Yes	
CURRENT	No	
CURSOR	Yes	
CURSOR-COL	Yes (C/S)	
CURSOR-COLOR	Yes (C/S)	
CURSOR-FRAME-WIDTH	Yes (C/S)	

CURSOR-ROW	Yes (C/S)
CURSOR-X	Yes (C/S)
CURSOR-Y	Yes (C/S)
CUSTOM-PRINT-TEMPLATE	Yes (C/S)
CYCLE	Yes (C/S)
CYL-INDEX	Yes (C/S)
CYL-OVERFLOW	Yes (C/S)
DASHED	Yes (C/S)
DATA	Yes
DATA-COLUMNS	Yes (C/S)
DATA-POINTER	No
DATA-TYPES	Yes (C/S)
DATE	Yes
DATE-COMPILED	Yes (C/S)
DATE-ENTRY	Yes (C/S)
DATE-MODIFIED	Yes (C/S)
DATE-WRITTEN	Yes (C/S)
DAY	Yes
DAY-OF-WEEK	Yes
DE	Yes
DEBUGGING	Yes
DECIMAL-POINT	Yes
DECLARATIVES	Yes
DEFAULT	Yes
DEFAULT-BUTTON	Yes (C/S)
DEFAULT-FONT	Yes
DELETE	Yes
DELIMITED	Yes
DELIMITER	Yes
DEPENDING	Yes
DESCENDING	Yes
DESTINATION	Yes
DESTROY	Yes
DETAIL	Yes
DISABLE	Yes
DISC	Yes (C/S)
DISK	Yes (C/S)
DISP	Yes (C/S)
DISPLAY	Yes
DISPLAY-COLUMNS	Yes (C/S)
DISPLAY-FORMAT	Yes (C/S)
DIVIDE	Yes
DIVIDER-COLOR	Yes (C/S)
DIVIDERS	Yes (C/S)
DIVISION	Yes
DOTDASH	Yes (C/S)

DOTTED	Yes (C/S)	
DOUBLE	Yes	FLOAT-LONG
DOWN	Yes	
DRAG-COLOR	Yes (C/S)	
DROP-DOWN	Yes (C/S)	
DROP-LIST	Yes (C/S)	
DUPLICATES	Yes	
DYNAMIC	Yes	
EBCDIC	Yes (C/S)	
EC	Yes	
ECHO	Yes	
EDITING	No	
EGI	Yes	
ELEMENT	Yes (C/S)	
ELSE	Yes	
EMI	Yes	
EMPTY-CHECK	Yes	REQUIRED
ENABLE	Yes	
ENCODING	Yes (C/S)	
ENCRYPTION	Yes (C/S)	
END	Yes	
END-ACCEPT	Yes	
END-ADD	Yes	
END-CALL	Yes	
END-CHAIN	No	
END-COLOR	Yes (C/S)	
END-COMPUTE	Yes	
END-DELETE	Yes	
END-DISPLAY	Yes	
END-DIVIDE	Yes	
END-EVALUATE	Yes	
END-IF	Yes	
END-JSON	Yes	
END-MODIFY	Yes (C/S)	
END-MULTIPLY	Yes	
END-OF-PAGE	Yes	EOP
END-PERFORM	Yes	
END-READ	Yes	
END-RECEIVE	Yes	
END-RETURN	Yes	
END-REWRITE	Yes	
END-SEARCH	Yes	
END-SEND	Yes	
END-START	Yes	
END-STRING	Yes	
END-SUBTRACT	Yes	

END-UNSTRING	Yes	
END-WRITE	Yes	
END-XML	Yes	
ENGRAVED	Yes (C/S)	
ENSURE-VISIBLE	Yes (C/S)	
ENTRY	Yes	
ENTRY-CONVENTION	Yes (C/S)	
ENTRY-FIELD	Yes (C/S)	
ENTRY-REASON	Yes (C/S)	
ENVIRONMENT	Yes	
ENVIRONMENT-NAME	Yes	
ENVIRONMENT-VALUE	Yes	
EO	No	
EOL	Yes (C/S)	
EOP	Yes	END-OF-PAGE
EOS	Yes (C/S)	
EQUAL	Yes	EQUALS
EQUALS	Yes	EQUAL
ERASE	Yes (C/S)	
ERROR	Yes	
ESCAPE	Yes	
ESCAPE-BUTTON	Yes (C/S)	
ESI	Yes	
EVALUATE	Yes	
EVENT	Yes	
EVENT-LIST	Yes (C/S)	
EVERY	Yes (C/S)	
EXCEPTION	Yes	
EXCEPTION-OBJECT	No	
EXCEPTION-VALUE	Yes (C/S)	
EXCLUSIVE	Yes	
EXCLUSIVE-OR	No	
EXHIBIT	Yes	
EXIT	Yes	
EXPAND	Yes (C/S)	
EXPANDS	No (C/S)	
EXTEND	Yes	
EXTENDED-SEARCH	Yes (C/S)	
EXTERN	Yes (C/S)	
EXTERNAL	Yes	
EXTERNAL-FORM	Yes	
F	Yes (C/S)	
FACTORY	No	
FALSE	Yes	
FD	Yes	
FH--FCD	Yes (C/S)	

FH--KEYDEF	Yes (C/S)	
FILE	Yes	
FILE-CONTROL	Yes	
FILE-ID	Yes	
FILE-LIMIT	Yes (C/S)	
FILE-LIMITS	Yes (C/S)	
FILE-NAME	Yes (C/S)	
FILE-POS	Yes (C/S)	
FILL-COLOR	Yes (C/S)	
FILL-COLOR2	Yes (C/S)	
FILL-PERCENT	Yes (C/S)	
FILLER	Yes	
FINAL	Yes	
FINALLY	No	
FINISH-REASON	Yes (C/S)	
FIRST	Yes	
FIXED	Yes	
FIXED-FONT	Yes	
FIXED-WIDTH	Yes (C/S)	
FLAT	Yes (C/S)	
FLAT-BUTTONS	Yes (C/S)	
FLOAT	Yes	FLOAT-SHORT
FLOAT-BINARY-128	No	
FLOAT-BINARY-32	No	
FLOAT-BINARY-64	No	
FLOAT-DECIMAL-16	Yes	
FLOAT-DECIMAL-34	Yes	
FLOAT-EXTENDED	No	
FLOAT-INFINITY	No	
FLOAT-LONG	Yes	DOUBLE
FLOAT-NOT-A-NUMBER	No (C/S)	
FLOAT-SHORT	Yes	FLOAT
FLOATING	Yes	
FONT	Yes	
FOOTING	Yes	
FOR	Yes	
FOREGROUND-COLOR	Yes (C/S)	FOREGROUND-COLOUR
FOREGROUND-COLOUR	Yes	FOREGROUND-COLOR
FOREVER	Yes (C/S)	
FORMAT	No	
FRAME	Yes (C/S)	
FRAMED	Yes (C/S)	
FREE	Yes	
FROM	Yes	
FULL	Yes (C/S)	LENGTH-CHECK
FULL-HEIGHT	Yes (C/S)	

FUNCTION	Yes	
FUNCTION-ID	Yes	
FUNCTION-POINTER	No	
GENERATE	Yes	
GET	No	
GIVING	Yes	
GLOBAL	Yes	
GO	Yes	
GO-BACK	Yes (C/S)	
GO-FORWARD	Yes (C/S)	
GO-HOME	Yes (C/S)	
GO-SEARCH	Yes (C/S)	
GOBACK	Yes	
GRAPHICAL	Yes (C/S)	
GREATER	Yes	
GRID	Yes (C/S)	
GROUP	Yes	
GROUP-USAGE	No	
GROUP-VALUE	Yes (C/S)	
HANDLE	Yes	
HAS-CHILDREN	Yes (C/S)	
HEADING	Yes	
HEADING-COLOR	Yes (C/S)	
HEADING-DIVIDER-COLOR	Yes (C/S)	
HEADING-FONT	Yes (C/S)	
HEAVY	Yes (C/S)	
HEIGHT-IN-CELLS	Yes (C/S)	
HEX	No (C/S)	
HIDDEN-DATA	Yes (C/S)	
HIGH-COLOR	Yes (C/S)	
HIGH-VALUE	Yes	HIGH-VALUES
HIGH-VALUES	Yes	HIGH-VALUE
HIGHLIGHT	Yes (C/S)	
HOT-TRACK	Yes (C/S)	
HSCROLL	Yes (C/S)	
HSCROLL-POS	Yes (C/S)	
I-O	Yes	
I-O-CONTROL	Yes	
ICON	Yes (C/S)	
ID	Yes	
IDENTIFICATION	Yes	
IDENTIFIED	Yes	
IF	Yes	
IGNORE	Yes	
IGNORING	Yes (C/S)	
IMPLEMENTS	No (C/S)	

IN	Yes	
INDEPENDENT	Yes (C/S)	
INDEX	Yes	
INDEXED	Yes	
INDICATE	Yes	
INHERITS	No	
INITIAL	Yes	
INITIALISE	Yes	INITIALIZE
INITIALISED	Yes	INITIALIZED
INITIALIZE	Yes	INITIALISE
INITIALIZED	Yes (C/S)	INITIALISED
INITIATE	Yes	
INPUT	Yes	
INPUT-OUTPUT	Yes	
INQUIRE	Yes	
INSERT-ROWS	Yes (C/S)	
INSERTION-INDEX	Yes (C/S)	
INSPECT	Yes	
INSTALLATION	Yes (C/S)	
INTERFACE	No	
INTERFACE-ID	No	
INTERMEDIATE	Yes (C/S)	
INTO	Yes	
INTRINSIC	Yes (C/S)	
INVALID	Yes	
INVOKE	No	
IS	Yes	
ITEM	Yes (C/S)	
ITEM-TEXT	Yes (C/S)	
ITEM-TO-ADD	Yes (C/S)	
ITEM-TO-DELETE	Yes (C/S)	
ITEM-TO-EMPTY	Yes (C/S)	
ITEM-VALUE	Yes (C/S)	
JSON	Yes	
JUST	Yes	JUSTIFIED
JUSTIFIED	Yes	JUST
KEPT	Yes	
KEY	Yes	
KEYBOARD	Yes (C/S)	
LABEL	Yes	
LABEL-OFFSET	Yes (C/S)	
LARGE-FONT	Yes	
LARGE-OFFSET	Yes (C/S)	
LAST	Yes	
LAST-ROW	Yes (C/S)	
LAYOUT-DATA	Yes (C/S)	

LAYOUT-MANAGER	Yes	
LC_ALL	No (C/S)	
LC_COLLATE	No (C/S)	
LC_CTYPE	No (C/S)	
LC_MESSAGES	No (C/S)	
LC_MONETARY	No (C/S)	
LC_NUMERIC	No (C/S)	
LC_TIME	No (C/S)	
LEADING	Yes	
LEADING-SHIFT	Yes (C/S)	
LEAVE	Yes (C/S)	
LEFT	Yes	
LEFT-JUSTIFY	No	
LEFT-TEXT	Yes (C/S)	
LEFTLINE	Yes	
LENGTH	Yes	
LENGTH-CHECK	Yes	FULL
LESS	Yes	
LIKE	Yes	
LIMIT	Yes	
LIMITS	Yes	
LINAGE	Yes	
LINAGE-COUNTER	Yes	
LINE	Yes	
LINE-COUNTER	Yes	
LINE-SEQUENTIAL	Yes (C/S)	
LINES	Yes	
LINES-AT-ROOT	Yes (C/S)	
LINKAGE	Yes	
LIST-BOX	Yes (C/S)	
LM-RESIZE	Yes	
LOC	Yes (C/S)	
LOCAL-STORAGE	Yes	
LOCALE	Yes	
LOCATION	No	
LOCK	Yes	
LOCK-HOLDING	Yes (C/S)	
LONG-DATE	Yes (C/S)	
LOW-COLOR	Yes (C/S)	
LOW-VALUE	Yes	LOW-VALUES
LOW-VALUES	Yes	LOW-VALUE
LOWER	Yes (C/S)	
LOWERED	Yes (C/S)	
LOWLIGHT	Yes (C/S)	
MAGNETIC-TAPE	Yes (C/S)	
MANUAL	Yes	

MASS-UPDATE	Yes (C/S)
MASTER-INDEX	Yes (C/S)
MAX-LINES	Yes (C/S)
MAX-PROGRESS	Yes (C/S)
MAX-TEXT	Yes (C/S)
MAX-VAL	Yes (C/S)
MEDIUM-FONT	Yes
MEMORY	Yes (C/S)
MENU	Yes
MERGE	Yes
MESSAGE	Yes
MESSAGE-TAG	No
METHOD	No
METHOD-ID	No
MIN-VAL	Yes (C/S)
MINUS	Yes
MODE	Yes
MODIFY	Yes
MODULES	Yes (C/S)
MOVE	Yes
MULTILINE	Yes (C/S)
MULTIPLE	Yes
MULTIPLY	Yes
NAME	Yes (C/S)
NAMED	Yes (C/S)
NAMESPACE	Yes (C/S)
NAMESPACE-PREFIX	Yes (C/S)
NAT	No
NATIONAL	Yes
NATIONAL-EDITED	Yes
NATIVE	Yes
NAVIGATE-URL	Yes (C/S)
NEAREST-AWAY-FROM-ZERO	Yes (C/S)
NEAREST-EVEN	Yes (C/S)
NEAREST-TOWARD-ZERO	Yes (C/S)
NEGATIVE	Yes
NESTED	Yes
NEW	Yes
NEXT	Yes
NEXT-ITEM	Yes (C/S)
NO	Yes
NO-AUTO-DEFAULT	Yes (C/S)
NO-AUTOSEL	Yes (C/S)
NO-BOX	Yes (C/S)
NO-DIVIDERS	Yes (C/S)
NO-ECHO	Yes

NO-F4	Yes (C/S)	
NO-FOCUS	Yes (C/S)	
NO-GROUP-TAB	Yes (C/S)	
NO-KEY-LETTER	Yes (C/S)	
NO-SEARCH	Yes (C/S)	
NO-UPDOWN	Yes (C/S)	
NOMINAL	Yes (C/S)	
NONE	No (C/S)	
NONNUMERIC	Yes (C/S)	
NORMAL	Yes (C/S)	
NOT	Yes	
NOTAB	Yes (C/S)	
NOTHING	Yes	
NOTIFY	Yes (C/S)	
NOTIFY-CHANGE	Yes (C/S)	
NOTIFY-DBLCLICK	Yes (C/S)	
NOTIFY-SELCHANGE	Yes (C/S)	
NULL	Yes	NULLS
NULLS	Yes	NULL
NUM-COL-HEADINGS	Yes (C/S)	
NUM-ROWS	Yes (C/S)	
NUMBER	Yes	
NUMBERS	Yes	
NUMERIC	Yes	
NUMERIC-EDITED	Yes	
OBJECT	Yes	
OBJECT-COMPUTER	Yes	
OBJECT-REFERENCE	No	
OCCURS	Yes	
OF	Yes	
OFF	Yes	
OK-BUTTON	Yes (C/S)	
OMITTED	Yes	
ON	Yes	
ONLY	Yes	
OPEN	Yes	
OPTIONAL	Yes	
OPTIONS	Yes	
OR	Yes	
ORDER	Yes	
ORGANISATION	Yes	ORGANIZATION
ORGANIZATION	Yes	ORGANISATION
OTHER	Yes	
OTHERS	Yes (C/S)	
OUTPUT	Yes	
OVERFLOW	Yes	

OVERLAP-LEFT	Yes (C/S)	OVERLAP-TOP
OVERLAP-TOP	Yes (C/S)	OVERLAP-LEFT
OVERLINE	Yes	
OVERRIDE	No	
PACKED-DECIMAL	Yes	
PADDING	Yes	
PAGE	Yes	
PAGE-COUNTER	Yes	
PAGE-SETUP	Yes (C/S)	
PAGED	Yes (C/S)	
PARAGRAPH	Yes (C/S)	
PARENT	Yes (C/S)	
PARSE	Yes (C/S)	
PASCAL	Yes (C/S)	
PASSWORD	Yes (C/S)	
PERFORM	Yes	
PERMANENT	Yes (C/S)	
PF	Yes	
PH	Yes	
PHYSICAL	Yes	
PIC	Yes	PICTURE
PICTURE	Yes	PIC
PIXEL	Yes (C/S)	PIXELS
PIXELS	Yes	PIXEL
PLACEMENT	Yes (C/S)	
PLUS	Yes	
POINTER	Yes	
POP-UP	Yes (C/S)	
POS	Yes	
POSITION	Yes	
POSITION-SHIFT	Yes (C/S)	
POSITIVE	Yes	
PREFIXED	No (C/S)	
PRESENT	Yes	
PREVIOUS	Yes (C/S)	
PRINT	Yes (C/S)	
PRINT-NO-PROMPT	Yes (C/S)	
PRINT-PREVIEW	Yes (C/S)	
PRINTER	Yes (C/S)	
PRINTER-1	Yes (C/S)	
PRINTING	Yes	
PRIORITY	Yes	
PROCEDURE	Yes	
PROCEDURE-POINTER	Yes	PROGRAM-POINTER
PROCEDURES	Yes	
PROCEED	Yes	

PROCESSING	Yes (C/S)	
PROGRAM	Yes	
PROGRAM-ID	Yes	
PROGRAM-POINTER	Yes	PROCEDURE-POINTER
PROGRESS	Yes (C/S)	
PROHIBITED	Yes (C/S)	
PROMPT	Yes	
PROPERTIES	Yes (C/S)	
PROPERTY	Yes	
PROTECTED	Yes (C/S)	
PROTOTYPE	No	
PURGE	Yes	
PUSH-BUTTON	Yes (C/S)	
QUERY-INDEX	Yes (C/S)	
QUEUE	Yes	
QUOTE	Yes	QUOTES
QUOTES	Yes	QUOTE
RADIO-BUTTON	Yes (C/S)	
RAISE	Yes	
RAISED	Yes (C/S)	
RAISING	No	
RANDOM	Yes	
RD	Yes	
READ	Yes	
READ-ONLY	Yes (C/S)	
READERS	Yes (C/S)	
RECEIVE	Yes	
RECEIVED	Yes	
RECORD	Yes	
RECORD-DATA	Yes (C/S)	
RECORD-OVERFLOW	Yes (C/S)	
RECORD-TO-ADD	Yes (C/S)	
RECORD-TO-DELETE	Yes (C/S)	
RECORDING	Yes	
RECORDS	Yes	
RECURSIVE	Yes (C/S)	
REDEFINES	Yes	
REEL	Yes	
REFERENCE	Yes	
REFERENCES	Yes	
REFRESH	Yes (C/S)	
REGION-COLOR	Yes (C/S)	
RELATION	No (C/S)	
RELATIVE	Yes	
RELEASE	Yes	
REMAINDER	Yes	

REMARKS	Yes (C/S)	
REMOVAL	Yes	
RENAMES	Yes	
REORG-CRITERIA	Yes (C/S)	
REPEATED	Yes	
REPLACE	Yes	
REPLACING	Yes	
REPORT	Yes	
REPORTING	Yes	
REPORTS	Yes	
REPOSITORY	Yes	
REQUIRED	Yes (C/S)	EMPTY-CHECK
REREAD	Yes (C/S)	
RERUN	Yes (C/S)	
RESERVE	Yes	
RESET	Yes	
RESET-GRID	Yes (C/S)	
RESET-LIST	Yes (C/S)	
RESET-TABS	Yes (C/S)	
RESUME	No	
RETRY	Yes	
RETURN	Yes	
RETURNING	Yes	
REVERSE	Yes	
REVERSE-VIDEO	Yes (C/S)	
REVERSED	Yes	
REWIND	Yes	
REWRITE	Yes	
RF	Yes	
RH	Yes	
RIGHT	Yes	
RIGHT-ALIGN	Yes (C/S)	
RIGHT-JUSTIFY	No	
RIMMED	Yes (C/S)	
ROLLBACK	Yes	
ROUNDED	Yes	
ROUNDING	Yes (C/S)	
ROW-COLOR	Yes (C/S)	
ROW-COLOR-PATTERN	Yes (C/S)	
ROW-DIVIDERS	Yes (C/S)	
ROW-FONT	Yes (C/S)	
ROW-HEADINGS	Yes (C/S)	
ROW-PROTECTION	Yes (C/S)	
RUN	Yes	
S	Yes (C/S)	
SAME	Yes	

SAVE-AS	Yes (C/S)
SAVE-AS-NO-PROMPT	Yes (C/S)
SCREEN	Yes
SCROLL	Yes (C/S)
SCROLL-BAR	Yes (C/S)
SD	Yes
SEARCH	Yes
SEARCH-OPTIONS	Yes (C/S)
SEARCH-TEXT	Yes (C/S)
SECONDS	Yes (C/S)
SECTION	Yes
SECURE	Yes (C/S)
SECURITY	Yes (C/S)
SEGMENT	Yes
SEGMENT-LIMIT	Yes
SELECT	Yes
SELECT-ALL	Yes (C/S)
SELECTION-INDEX	Yes (C/S)
SELECTION-TEXT	Yes (C/S)
SELF	No
SELF-ACT	Yes (C/S)
SEND	Yes
SENTENCE	Yes
SEPARATE	Yes
SEPARATION	Yes (C/S)
SEQUENCE	Yes
SEQUENTIAL	Yes
SET	Yes
SHADING	Yes (C/S)
SHADOW	Yes (C/S)
SHARING	Yes
SHORT-DATE	Yes (C/S)
SHOW-LINES	Yes (C/S)
SHOW-NONE	Yes (C/S)
SHOW-SEL-ALWAYS	Yes (C/S)
SIGN	Yes
SIGNED	Yes
SIGNED-INT	Yes
SIGNED-LONG	Yes
SIGNED-SHORT	Yes
SIZE	Yes
SMALL-FONT	Yes
SORT	Yes
SORT-MERGE	Yes
SORT-ORDER	Yes (C/S)
SOURCE	Yes

SOURCE-COMPUTER	Yes	
SOURCES	No	
SPACE	Yes	SPACES
SPACE-FILL	No	
SPACES	Yes	SPACE
SPECIAL-NAMES	Yes	
SPINNER	Yes (C/S)	
SQUARE	Yes (C/S)	
STACK	No (C/S)	
STANDARD	Yes	
STANDARD-1	Yes	
STANDARD-2	Yes	
STANDARD-BINARY	Yes (C/S)	
STANDARD-DECIMAL	Yes (C/S)	
START	Yes	
START-X	Yes (C/S)	
START-Y	Yes (C/S)	
STATEMENT	No (C/S)	
STATIC	Yes (C/S)	
STATIC-LIST	Yes (C/S)	
STATUS	Yes	
STATUS-BAR	Yes (C/S)	
STATUS-TEXT	Yes (C/S)	
STDCALL	Yes (C/S)	
STEP	Yes (C/S)	
STOP	Yes	
STRING	Yes	
STRONG	No (C/S)	
STYLE	Yes (C/S)	
SUB-QUEUE-1	Yes	
SUB-QUEUE-2	Yes	
SUB-QUEUE-3	Yes	
SUBTRACT	Yes	
SUBWINDOW	Yes	
SUM	Yes	
SUPER	No	
SUPPRESS	Yes	
SYMBOL	No (C/S)	
SYMBOLIC	Yes	
SYNC	Yes	SYNCHRONISED, SYNCHRONIZED
SYNCHRONISED	Yes	SYNC, SYNCHRONIZED
SYNCHRONIZED	Yes	SYNC, SYNCHRONISED
SYSTEM-DEFAULT	Yes	
SYSTEM-INFO	Yes (C/S)	
SYSTEM-OFFSET	Yes	
TAB	Yes (C/S)	

TAB-TO-ADD	Yes (C/S)	
TAB-TO-DELETE	Yes (C/S)	
TABLE	Yes	
TALLYING	Yes	
TAPE	Yes (C/S)	
TEMPORARY	Yes (C/S)	
TERMINAL-INFO	Yes (C/S)	
TERMINATE	Yes	
TERMINATION-VALUE	Yes (C/S)	
TEST	Yes	
TEXT	Yes	
THAN	Yes	
THEN	Yes	
THREAD	Yes	
THREADS	Yes	
THROUGH	Yes	THRU
THRU	Yes	THROUGH
THUMB-POSITION	Yes (C/S)	
TILED-HEADINGS	Yes (C/S)	
TIME	Yes	
TIME-OUT	Yes (C/S)	TIMEOUT
TIMEOUT	Yes	TIME-OUT
TIMES	Yes	
TITLE	Yes (C/S)	
TITLE-POSITION	Yes (C/S)	
TO	Yes	
TOP	Yes	
TOP-LEVEL	No (C/S)	
TOWARD-GREATER	Yes (C/S)	
TOWARD-LESSER	Yes (C/S)	
TRACK	Yes (C/S)	
TRACK-AREA	Yes (C/S)	
TRACK-LIMIT	Yes (C/S)	
TRACKS	Yes (C/S)	
TRADITIONAL-FONT	Yes	
TRAILING	Yes	
TRAILING-SHIFT	Yes (C/S)	
TRAILING-SIGN	No	
TRANSFORM	Yes	
TRANSPARENT	Yes (C/S)	
TREE-VIEW	Yes (C/S)	
TRUE	Yes	
TRUNCATION	Yes (C/S)	
TYPE	Yes	
TYPEDEF	Yes	
U	Yes (C/S)	

UCS-4	Yes (C/S)	
UNBOUNDED	Yes (C/S)	
UNDERLINE	Yes (C/S)	
UNFRAMED	Yes (C/S)	
UNIT	Yes	
UNIVERSAL	No	
UNLOCK	Yes	
UNSIGNED	Yes	
UNSIGNED-INT	Yes	
UNSIGNED-LONG	Yes	
UNSIGNED-SHORT	Yes	
UNSORTED	Yes (C/S)	
UNSTRING	Yes	
UNTIL	Yes	
UP	Yes	
UPDATE	Yes	
UPDATERS	Yes (C/S)	
UPON	Yes	
UPPER	Yes (C/S)	
USAGE	Yes	
USE	Yes	
USE-ALT	Yes (C/S)	
USE-RETURN	Yes (C/S)	
USE-TAB	Yes (C/S)	
USER	Yes (C/S)	
USER-DEFAULT	Yes	
USING	Yes	
UTF-16	Yes (C/S)	
UTF-8	Yes (C/S)	
V	Yes (C/S)	
VAL-STATUS	No	
VALID	No	
VALIDATE	Yes	
VALIDATE-STATUS	No	
VALIDATING	Yes (C/S)	
VALUE	Yes	VALUES
VALUE-FORMAT	Yes (C/S)	
VALUES	Yes	VALUE
VARIABLE	Yes (C/S)	
VARIANT	Yes	
VARYING	Yes	
VERTICAL	Yes (C/S)	
VERY-HEAVY	Yes (C/S)	
VIRTUAL-WIDTH	Yes (C/S)	
VOLATILE	Yes	
VPADDING	Yes (C/S)	

VSCROLL	Yes (C/S)	
VSCROLL-BAR	Yes (C/S)	
VSCROLL-POS	Yes (C/S)	
VTOP	Yes (C/S)	
WAIT	Yes	
WEB-BROWSER	Yes (C/S)	
WHEN	Yes	
WIDTH	Yes (C/S)	
WIDTH-IN-CELLS	Yes (C/S)	
WINDOW	Yes	
WITH	Yes	
WORDS	Yes	
WORKING-STORAGE	Yes	
WRAP	Yes (C/S)	
WRITE	Yes	
WRITE-ONLY	Yes (C/S)	
WRITE-VERIFY	Yes (C/S)	
WRITERS	Yes (C/S)	
X	Yes (C/S)	
XML	Yes	
XML-DECLARATION	Yes (C/S)	
XML-SCHEMA	Yes (C/S)	
XOR	No	
Y	Yes (C/S)	
YYYYDDD	Yes (C/S)	
YYYYMMDD	Yes (C/S)	
ZERO	Yes	ZEROES, ZEROS
ZERO-FILL	No (C/S)	
ZEROES	Yes	ZERO, ZEROS
ZEROS	Yes	ZERO, ZEROES

Extra (obsolete) context sensitive words

AUTHOR, DATE-COMPILED, DATE-MODIFIED, DATE-WRITTEN, INSTALLATION, REMARKS, SECURITY

Internal registers

Register	Implemented	Definition
'ADDRESS OF' phrase	Yes	USAGE POINTER
COB-CRT-STATUS	Yes	PICTURE 9(4) USAGE DISPLAY VALUE ZERO
DEBUG-ITEM	Yes	PICTURE X(n) USAGE DISPLAY
'LENGTH OF' phrase	Yes	CONSTANT USAGE BINARY-LONG
NUMBER-OF-CALL-PARAMETERS	Yes	USAGE BINARY-LONG

RETURN-CODE	Yes	GLOBAL USAGE BINARY-LONG VALUE ZERO
SORT-RETURN	Yes	GLOBAL USAGE BINARY-LONG VALUE ZERO
TALLY	Yes	GLOBAL PICTURE 9(5) USAGE BINARY VALUE ZERO
WHEN-COMPILED	Yes	CONSTANT PICTURE X(16) USAGE DISPLAY
XML-CODE	Yes	GLOBAL PICTURE S9(9) USAGE BINARY VALUE 0
XML-EVENT	Yes GLOBAL USAGE DISPLAY PICTURE X(30) VALUE SPACE	
XML-INFORMATION	Yes GLOBAL PICTURE S9(9) USAGE BINARY VALUE 0	
XML-NAMESPACE	Yes GLOBAL PIC X ANY LENGTH	
XML-NAMESPACE-PREFIX	Yes GLOBAL PIC X ANY LENGTH	
XML-NNAMESPACE	Yes GLOBAL PIC N ANY LENGTH	
XML-NNAMESPACE-PREFIX	Yes GLOBAL PIC N ANY LENGTH	
XML-NTEXT	Yes GLOBAL PIC N ANY LENGTH	
XML-TEXT	Yes GLOBAL PIC X ANY LENGTH	
JSON-CODE	Yes	GLOBAL PICTURE S9(9) USAGE BINARY VALUE 0
JSON-STATUS	Yes	GLOBAL PICTURE S9(9) USAGE BINARY VALUE 0

Appendix C2 - Intrinsic Functions

The following list of intrinsic functions was extracted from `cobc --list-intrinsics` and shows the names of the available functions, an implementation note and the number of parameters.

Intrinsic Function	Implemented	Parameters
ABS	Yes	1
ACOS	Yes	1
ANNUITY	Yes	2
ASIN	Yes	1
ATAN	Yes	1
BASECONVERT	No	3
BIT-OF	Yes	1
BIT-TO-CHAR	Yes	1
BOOLEAN-OF-INTEGER	No	2
BYTE-LENGTH	Yes	1 - 2
CHAR	Yes	1
CHAR-NATIONAL	No	1
COMBINED-DATETIME	Yes	2
CONCAT	Yes	Unlimited
CONCATENATE	Yes	Unlimited
CONTENT-LENGTH	Yes	1
CONTENT-OF	Yes	1 - 2
CONVERT	No	3
COS	Yes	1
CURRENCY-SYMBOL	Yes	0
CURRENT-DATE	Yes	0
DATE-OF-INTEGER	Yes	1
DATE-TO-YYYYMMDD	Yes	1 - 3
DAY-OF-INTEGER	Yes	1
DAY-TO-YYYYDDD	Yes	1 - 3
DISPLAY-OF	No	1 - 2
E	Yes	0
EXCEPTION-FILE	Yes	0
EXCEPTION-FILE-N	No	0
EXCEPTION-LOCATION	Yes	0
EXCEPTION-LOCATION-N	No	0
EXCEPTION-STATEMENT	Yes	0
EXCEPTION-STATUS	Yes	0
EXP	Yes	1
EXP10	Yes	1
FACTORIAL	Yes	1
FIND-STRING	No	7 - 2
FORMATTED-CURRENT-DATE	Yes	1
FORMATTED-DATE	Yes	2

FORMATTED-DATETIME	Yes	4 - 5
FORMATTED-TIME	Yes	3 - 4
FRACTION-PART	Yes	1
HEX-OF	Yes	1
HEX-TO-CHAR	Yes	1
HIGHEST-ALGEBRAIC	Yes	1
INTEGER	Yes	1
INTEGER-OF-BOOLEAN	No	1
INTEGER-OF-DATE	Yes	1
INTEGER-OF-DAY	Yes	1
INTEGER-OF-FORMATTED-DATE	Yes	2
INTEGER-PART	Yes	1
LENGTH	Yes	1 - 2
LENGTH-AN	Yes	1
LOCALE-COMPARE Yes 2 - 3		
LOCALE-DATE	Yes	1 - 2
LOCALE-TIME	Yes	1 - 2
LOCALE-TIME-FROM-SECONDS	Yes	1 - 2
LOG	Yes	1
LOG10	Yes	1
LOWER-CASE	Yes	1
LOWEST-ALGEBRAIC	Yes	1
MAX	Yes	Unlimited
MEAN	Yes	Unlimited
MEDIAN	Yes	Unlimited
MIDRANGE	Yes	Unlimited
MIN	Yes	Unlimited
MOD	Yes	2
MODULE-CALLER-ID	Yes	0
MODULE-DATE	Yes	0
MODULE-FORMATTED-DATE	Yes	0
MODULE-ID	Yes	0
MODULE-NAME	No	1
MODULE-PATH	Yes	0
MODULE-SOURCE	Yes	0
MODULE-TIME	Yes	0
MONETARY-DECIMAL-POINT	Yes	0
MONETARY-THOUSANDS-SEPARATOR	Yes	0
NATIONAL-OF	No	1 - 2
NUMERIC-DECIMAL-POINT	Yes	0
NUMERIC-THOUSANDS-SEPARATOR	Yes	0
NUMVAL	Yes	1
NUMVAL-C	Yes	2
NUMVAL-F	Yes	1
ORD	Yes	1
ORD-MAX	Yes	Unlimited

ORD-MIN	Yes	Unlimited
PI	Yes	0
PRESENT-VALUE	Yes	Unlimited
RANDOM	Yes	0 - 1
RANGE	Yes	Unlimited
REM	Yes	2
REVERSE	Yes	1
SECONDS-FROM-FORMATTED-TIME	Yes	2
SECONDS-PAST-MIDNIGHT	Yes	0
SIGN	Yes	1
SIN	Yes	1
SQRT	Yes	1
STANDARD-COMPARE	No	2 - 4
STANDARD-DEVIATION	Yes	Unlimited
STORED-CHAR-LENGTH	Yes	1
SUBSTITUTE	Yes	Unlimited
SUBSTITUTE-CASE	Yes	Unlimited
SUM	Yes	Unlimited
TAN	Yes	1
TEST-DATE-YYYYMMDD	Yes	1
TEST-DAY-YYYYDDD	Yes	1
TEST-FORMATTED-DATETIME	Yes	2
TEST-NUMVAL	Yes	1
TEST-NUMVAL-C	Yes	2
TEST-NUMVAL-F	Yes	1
TRIM	Yes	1 - 2
UPPER-CASE	Yes	1
VARIANCE	Yes	Unlimited
WHEN-COMPILED	Yes	0
YEAR-TO-YYYY	Yes	1 - 3

Appendix C3 - System routines

The following list of system routines was extracted from `cobc --list-system` and shows the names of the available system routines along with the number of parameters.

System routine	Parameters
SYSTEM	1
CBL_AND	3
CBL_ALARM_SOUND	0
CBL_BELL_SOUND	0
CBL_CHANGE_DIR	1
CBL_CHECK_FILE_EXIST	2
CBL_CLOSE_FILE	1
CBL_COPY_FILE	2
CBL_CREATE_DIR	1
CBL_CREATE_FILE	5
CBL_DELETE_DIR	1
CBL_DELETE_FILE	1
CBL_EQ	3
CBL_ERROR_PROC	2
CBL_EXIT_PROC	2
CBL_FLUSH_FILE	1
CBL_GET_CSR_POS	1
CBL_GET_CURRENT_DIR	3
CBL_GET_SCR_SIZE	2
CBL_IMP	3
CBL_NIMP	3
CBL_NOR	3
CBL_NOT	2
CBL_OPEN_FILE	5
CBL_OR	3
CBL_READ_FILE	5
CBL_READ_KBD_CHAR	1
CBL_RENAME_FILE	2
CBL_SET_CSR_POS	1
CBL_TOLOWER	2
CBL_TOUPPER	2
CBL_WRITE_FILE	5
CBL_XOR	3
CBL_GC_FORK	0
CBL_GC_GETOPT	6
CBL_GC_HOSTED	2
CBL_GC_NANOSLEEP	1
CBL_GC_PRINTABLE	1 - 2
CBL_GC_WAITPID	1
CBL_OC_GETOPT	6

CBL_OC_HOSTED	2
CBL_OC_NANOSLEEP	1
C$CALLEDBY	1
C$CHDIR	2
C$COPY	3
C$DELETE	2
C$FILEINFO	2
C$GETPID	0
C$JUSTIFY	1 - 2
C$MAKEDIR	1
C$NARG	1
C$PARAMSIZE	1
C$PRINTABLE	1 - 2
C$SLEEP	1
C$TOLOWER	2
C$TOUPPER	2
EXTFH	2
X"91"	3
X"E4"	0
X"E5"	0
X"F4"	2
X"F5"	2

Appendix C4 - System names

The following list of system names was extracted from `cobc --list-mnemonics` and shows the system names categorized by their type.

System names: device

SYSIN, SYSIPT, STDIN, SYSOUT, SYSLIST, SYSLST, SYSPCH, SYSPUNCH, STDOUT, PRINT, PRINTER, PRINTER-1, SYSERR, STDERR, CONSOLE, ALTERNATE-CONSOLE, ALTERNATE CONSOLE

System names: feature

C01, C02, C03, C04, C05, C06, C07, C08, C09, C10, C11, C12, S01, S02, S03, S04, S05, CSP, FORMFEED, TOP, CALL-CONVENTION

System names: switch

SWITCH-0, SWITCH-1, SWITCH-2, SWITCH-3, SWITCH-4, SWITCH-5, SWITCH-6, SWITCH-7, SWITCH-8, SWITCH-9, SWITCH-10, SWITCH-11, SWITCH-12, SWITCH-13, SWITCH-14, SWITCH-15, SWITCH-16, SWITCH-17, SWITCH-18, SWITCH-19, SWITCH-20, SWITCH-21, SWITCH-22, SWITCH-23, SWITCH-24, SWITCH-25, SWITCH-26, SWITCH-27, SWITCH-28, SWITCH-29, SWITCH-30, SWITCH-31, SWITCH-32, SWITCH-33, SWITCH-34, SWITCH-35, SWITCH-36

Appendix C5 - Exceptions

The following list of exceptions names was extracted from `cobc --list-exceptions` and shows the names by type.

```
Exception Name
EC-ALL
  EC-ARGUMENT
    EC-ARGUMENT-FUNCTION (f)
    EC-ARGUMENT-IMP
  EC-BOUND
    EC-BOUND-FUNC-RET-VALUE
    EC-BOUND-IMP
    EC-BOUND-ODO (f)
    EC-BOUND-OVERFLOW (f)
    EC-BOUND-PTR (f)
    EC-BOUND-REF-MOD (f)
    EC-BOUND-SET (f)
    EC-BOUND-SUBSCRIPT (f)
    EC-BOUND-TABLE-LIMIT (f)
  EC-CONTINUE
    EC-CONTINUE-IMP
    EC-CONTINUE-LESS-THAN-ZERO
  EC-DATA
    EC-DATA-CONVERSION
    EC-DATA-IMP
    EC-DATA-INCOMPATIBLE (f)
    EC-DATA-NOT-FINITE (f)
    EC-DATA-OVERFLOW (f)
    EC-DATA-PTR-NULL (f)
  EC-EXTERNAL
    EC-EXTERNAL-DATA-MISMATCH (f)
    EC-EXTERNAL-FILE-MISMATCH (f)
    EC-EXTERNAL-FORMAT-CONFLICT (f)
    EC-EXTERNAL-IMP
  EC-FLOW
    EC-FLOW-APPLY-COMMIT (f)
    EC-FLOW-COMMIT (f)
    EC-FLOW-GLOBAL-EXIT (f)
    EC-FLOW-GLOBAL-GOBACK (f)
    EC-FLOW-IMP
    EC-FLOW-RELEASE (f)
    EC-FLOW-REPORT (f)
    EC-FLOW-RETURN (f)
    EC-FLOW-ROLLBACK (f)
    EC-FLOW-SEARCH (f)
```

```
    EC-FLOW-USE (f)
  EC-FUNCTION
    EC-FUNCTION-ARG-OMITTED (f)
    EC-FUNCTION-IMP
    EC-FUNCTION-NOT-FOUND (f)
    EC-FUNCTION-PTR-INVALID (f)
    EC-FUNCTION-PTR-NULL (f)
  EC-I-O
    EC-I-O-AT-END
    EC-I-O-EOP
    EC-I-O-EOP-OVERFLOW
    EC-I-O-FILE-SHARING
    EC-I-O-IMP
    EC-I-O-INVALID-KEY
    EC-I-O-LINAGE (f)
    EC-I-O-LOGIC-ERROR (f)
    EC-I-O-PERMANENT-ERROR (f)
    EC-I-O-RECORD-CONTENT (f)
    EC-I-O-RECORD-OPERATION
    EC-I-O-RECORD-WARNING
  EC-IMP
    EC-IMP-ACCEPT
    EC-IMP-DISPLAY
    EC-IMP-UTC-UNKNOWN (f)
    EC-IMP-FEATURE-DISABLED
    EC-IMP-FEATURE-MISSING
  EC-LOCALE
    EC-LOCALE-IMP
    EC-LOCALE-INCOMPATIBLE
    EC-LOCALE-INVALID (f)
    EC-LOCALE-INVALID-PTR (f)
    EC-LOCALE-MISSING (f)
    EC-LOCALE-SIZE (f)
  EC-MCS
    EC-MCS-ABNORMAL-TERMINATION
    EC-MCS-IMP
    EC-MCS-INVALID-TAG
    EC-MCS-MESSAGE-LENGTH
    EC-MCS-NO-REQUESTER
    EC-MCS-NO-SERVER
    EC-MCS-NORMAL-TERMINATION
    EC-MCS-REQUESTOR-FAILED
  EC-OO
    EC-OO-ARG-OMITTED (f)
    EC-OO-CONFORMANCE (f)
    EC-OO-EXCEPTION (f)
```

```
    EC-OO-IMP
    EC-OO-METHOD (f)
    EC-OO-NULL (f)
    EC-OO-RESOURCE (f)
    EC-OO-UNIVERSAL (f)
  EC-ORDER
    EC-ORDER-IMP
    EC-ORDER-NOT-SUPPORTED (f)
  EC-OVERFLOW
    EC-OVERFLOW-IMP
    EC-OVERFLOW-STRING
    EC-OVERFLOW-UNSTRING
  EC-PROGRAM
    EC-PROGRAM-ARG-MISMATCH (f)
    EC-PROGRAM-ARG-OMITTED (f)
    EC-PROGRAM-CANCEL-ACTIVE (f)
    EC-PROGRAM-IMP
    EC-PROGRAM-NOT-FOUND (f)
    EC-PROGRAM-PTR-NULL (f)
    EC-PROGRAM-RECURSIVE-CALL (f)
    EC-PROGRAM-RESOURCES (f)
  EC-RAISING
    EC-RAISING-IMP
    EC-RAISING-NOT-SPECIFIED (f)
  EC-RANGE
    EC-RANGE-IMP
    EC-RANGE-INDEX (f)
    EC-RANGE-INSPECT-SIZE (f)
    EC-RANGE-INVALID
    EC-RANGE-PERFORM-VARYING (f)
    EC-RANGE-PTR (f)
    EC-RANGE-SEARCH-INDEX
    EC-RANGE-SEARCH-NO-MATCH
  EC-REPORT
    EC-REPORT-ACTIVE (f)
    EC-REPORT-COLUMN-OVERLAP (f)
    EC-REPORT-FILE-MODE (f)
    EC-REPORT-IMP
    EC-REPORT-INACTIVE (f)
    EC-REPORT-LINE-OVERLAP
    EC-REPORT-NOT-TERMINATED
    EC-REPORT-PAGE-LIMIT
    EC-REPORT-PAGE-WIDTH
    EC-REPORT-SUM-SIZE (f)
    EC-REPORT-VARYING (f)
  EC-SCREEN
```

```
    EC-SCREEN-FIELD-OVERLAP
    EC-SCREEN-IMP
    EC-SCREEN-ITEM-TRUNCATED
    EC-SCREEN-LINE-NUMBER
    EC-SCREEN-STARTING-COLUMN
  EC-SIZE
    EC-SIZE-ADDRESS (f)
    EC-SIZE-EXPONENTIATION (f)
    EC-SIZE-IMP
    EC-SIZE-OVERFLOW (f)
    EC-SIZE-TRUNCATION (f)
    EC-SIZE-UNDERFLOW (f)
    EC-SIZE-ZERO-DIVIDE (f)
  EC-SORT-MERGE
    EC-SORT-MERGE-ACTIVE (f)
    EC-SORT-MERGE-FILE-OPEN (f)
    EC-SORT-MERGE-IMP
    EC-SORT-MERGE-RELEASE (f)
    EC-SORT-MERGE-RETURN (f)
    EC-SORT-MERGE-SEQUENCE (f)
  EC-STORAGE
    EC-STORAGE-IMP
    EC-STORAGE-NOT-ALLOC
    EC-STORAGE-NOT-AVAIL
  EC-USER
  EC-VALIDATE
    EC-VALIDATE-CONTENT
    EC-VALIDATE-FORMAT
    EC-VALIDATE-IMP
    EC-VALIDATE-RELATION
    EC-VALIDATE-VARYING (f)
  EC-XML
    EC-XML-CODESET (f)
    EC-XML-CODESET-CONVERSION (f)
    EC-XML-COUNT (f)
    EC-XML-DOCUMENT-TYPE (f)
    EC-XML-IMPLICIT-CLOSE (f)
    EC-XML-INVALID (f)
    EC-XML-NAMESPACE (f)
    EC-XML-STACKED-OPEN (f)
    EC-XML-RANGE (f)
    EC-XML-IMP (f)
  EC-JSON
    EC-JSON-IMP (f)
```

End of Appendix C — Grouped Reserved Word List

Appendix D - GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
`https://fsf.org/`

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/licenses/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

End of Appendix D — GNU Free Documentation License

Appendix E - Summary of Document Changes

GnuCOBOL is an ever-evolving tool. While all reasonable attempts will be made to maintain the currency of the information in this document, neither the author of this document nor the authors of the GnuCOBOL software extend any warranties of any kind for this document or for the information contained therein.

8th Edition on release of v3.2 with some updates for 4.0+.

1. 2021.

   25/01 Updated version to v3.1.2 with date 25 Jan 21.
2. 29/01 Updated formats for OCCURS and now only using one file. Ditto for Quick Reference.
3. 7/02 Added procedure statement EXHIBIT, updated reserved words in Appendix C and removed same in Appendix B.
4. 4/03 Modified Exhibit points 1/2 to loose the blank lines.
5. 12/04 Updated text for CBL_SET_CSR_POS, repos EXHIBIT to 15B.
6. 13/05 Updated USAGE for BINARY=C-LONG. Updated version to v3.2 with date 13 May 2021.
7. 15/05 Updated EXHIBIT and renumbered paragraphs.
8. 24/08 Added detail for ACCEPT DAY-OF-WEEK. Changed descriptive for Display screen-data-name removing screen and adding position specs see bug #743 now cleared.
9. 30/08 Change Appendix F to separate sections F1 through F4 adding changes for v3.1.1 and v4. Added updates.texi after E.texi for latest document updates.
10. 5/09 Added Intrinsic functions BIT-OF, BIT-TO-CHAR, HEX-OF, HEX-TO-CHAR as 8.1.105 thru 108.but needs resorting of # in 8.1. Updated definition for C$SLEEP.
11. 12.09 Multi typo's in sections 2, 3.5, .14, 6.9.51, 8.1.64B (chgd 2 to V2, 12.2, spotted by Patrik - Thank you for the updates.
12. 17/10 Extra information regarding LINE SEQUENTIAL file processing in 2.1.8 which refers to 10.2.3.4 and 10.2.3.7.
13. 17/10 CDF PAGE directive changes to details. Likewise to the Quick Reference manual.
14. 21/11 DIVIDE statement minor changes (#786).
15. 23/11 DIVIDE changed as 7.8.13.1 INTO and 13.3 BY, must have been asleep.
16. 25/11 Updated example listings in 10.1.1. updated to current default.conf in 10.1.5 as out of date.
17. 23/12 Update to START verb to make KEY compulsory for ISAM.
18. 24/12 Update to START added missing sub clauses [WITH SIZE] | [WITH LENGTH].
19. 25/12 Update to sections 10.2.2 and 10.2.3 to clarify where program modules are loaded from i.e., directories pointed to by COB_LIBRARY-PATH and only if not found in PATH. Update to syntax diagran for START to clarify that SIZE and LENGTH keywords are mandatory words even if obvious. Updated description for COMBINED-DATETIME

20. 31/12 Update to section 6.8.1 change program names to Usage-Lengths and replace the source listing to be correct positioned on cc8 as well as giving o/p for 64bit Linux. Bug #799.
21. 2022.
22. 07/01 Remove references to PATH in 10.2.2 &.3 wrong.
23. 16/01 In 2.2.5 near bottom example of compute 5 / 2.5 + 7 * 2 - 1.15 had wrong result, should be 14.85. Clean up descriptions for USAGE COMP[UTATIONAL] & COMP etc in 6.9.49 and 7.8.1.2 accept from command-line.
24. 24/01 In 7.8.23 type fixed of elementary - remove '3' no vers change.
25. 17/02 In DISPLAY type 4 added OMITTED in place of identifier-1 to stop any screen content to change.
26. 07/03 IN 7.8.1.9, Accept EXCEPTION-STATUS these are two words i.e., EXCEPTION STATUS not one.
27. 27/03 In 3.6 A comment included to advise that >>DEFINE should be used for new programs, over >>SET as it is not in the Cobol Standards. Included in 10.1 -Wno-terminator to stop warnings when not using END-XXX (Also see -Wterminator).
28. 07/04 IN section 3 CDF commands I have changed starting point from CC8 to cc7 as highlighted by Chris Gabel.
29. 21/04 In section 8.1 updated BYTE-LENGTH amd LENGTH-AN where referencing 16 bit characters to use N(4) instead of X(4).
30. 23/04 In section 11.6.1.1, for Definitions of 'By Reference' and 'By Content' added missing 'by'. Spotted by Michael F Gleason, bug 828.
31. 03/05 In 8.1.82 & 83 for SUBSTITUTE and SUBSTITUE-CASE text on usage of variables, the need to use function TRIM is needed to remove leading and trailing spaces must be considered with example code block.
32. 15/05 In section 7.8.48 TRANSFORM added 'CHARACTERS' to the format it has no effect with or without.
33. 1/06 In section 5.1.3 Show example of REPOSITORY has only the last entry with a period. In keeping with SPECIAL-NAMES etc that only the last entry has the period / full stop terminator. Use extra space in 6.9.49 to show that the TO value in the FROM / TO values is NOT negative as only the From value is usually is.
34. 26/06 In 2.2.4 (+) should have been (-). Added Type (format 1) TYPEDEF, SAME AS in 6.9 and shifted down section numbers.
35. 29/06 Minor corrections to text for TYPEDEF, SAME AS & missing page reference. USAGE text addition.
36. 1/07 Adding XDF chapter 13. Early days - needs tidying up.
37. 6/07 In 6.9.53 added WHEN SET TO FALSE sub-clause in SYN-DD-VALUE-1.
38. 9/07 In 6.9.49 typo date should be data.
39. 10/07 Added functions CONCAT, CONTENT-LENGTH and CONTENT-OF.
40. 11/07 Adjusted and sorted functions at 8.1.

41. 12/07 In 2.1.2 added details regardingsize of a user defined word i.e., from 31 chars to 63 using the right -std. In 7.8.17 EXIT statement RETURNING|GIVING missing from EXIT PROGRAM and description details.
42. 13/07 For ENTRY added missing GO TO and other content.
43. 16/07 For GO TO ENTRY entry-name added.
44. 27/08 Updated context for >>SOURCE in section 3.7 for FIXED, FREE, VARIABLE and others, likewise for >>SET. Updated copyright for FSF.
45. 05/09 Added new clauses in 6.9 - Data Description Clauses for PG and QR.
46. 11/09 Added PICTURE option 1 bit or boolean in 6.9.33 with a warning about compiler version implementation from v3.1-rc1. In 6.9.46 last sentence changed 'four' to 'eight' for a double word binary number.
47. 26/09 Added Standard 85 EXAMINE verb at 7.8.15B.
48. 29/09 Added extra notes for CODE-SET referring to compiler version and the NEWS file.
49. 06/12 At 8.2.44 for offset addition to "The first byte of a file is byte offset 0" is "and MUST be preset to zero for first use".
50. 2023.
51. 02/01 At 7.8.1.5 Added ACCEPT from MICROSECOND-TIME (ACU. New commit [r4908] by sf-mensch.
52. 03/01 At 7.8.1.5 Added more information regarding TIME and MICROSECONDS-TIME and update for FORMATTED-CURRENT-DATE in 8.1.29.
53. 04/01 7.8.1.5 Diagram had missing TIME and text had 2 copies of it.
54. 06/01 6.8.1 01-Level Constants, added arithmetic-espression-1. Re-introduced Updates from ed. 8.
55. 15/01 8.1.33 FRACTION-PART Included text to specify picture must include preceding 'V' and provide better example.
56. 25/01 8.1.94 TEST-FORMATTED-DATETIME - Improved the text and provide an example of usage. Added extra info regarding Date/Time formats but may not apply to specific compiler version.

Index

"

" (Quote) 35

$

$ Directives 80

&

& (Literal Concatenation) 35

'

' (Apostrophe) 35

*

* (Multiplication) 42
* In Column 7 (Comment) 33
** (Exponentiation) 42
*> (Comment) 33

+

+ (Addition) 43
+ (Unary Sign Retention) 42

,

, (Punctuation) 37

-

- (Character in Words/Names) 10
- (Subtraction) 43
- (Unary Sign Reversal) 42
- In Column 7 (Continuation) 35
-b Compiler Switch 682
-conf Compiler Switch 649
-debug Compiler Switch 255, 449, 450, 537
-fdebugging-line Compiler Switch 89, 255
-ffold-call Compiler Switch 658
-ffold-copy Compiler Switch 71, 648
-fintrinsics Compiler Switch 421
-fintrinsics=ALL Compiler Switch 101
-fixed Compiler Switch 29, 71, 73
-foptional-file Compiler Switch 105
-free Compiler Switch 29, 71, 73
-fsyntax-extension Compiler Switch 99
-ftrace Compiler Switch 370, 372, 659
-ftraceall Compiler Switch .. 372, 449, 450, 537, 659
-g Compiler Switch 449, 450, 537
-I Compiler Switch 648
-m Compiler Switch 641, 656, 682
-o Compiler Switch 641, 656
-O Compiler Switch 678
-O2 Compiler Switch 678
-Os Compiler Switch 678
-Wobsolete Compiler Switch 86
-x Compiler Switch 641, 656, 682, 721

.

. (Punctuation) 37

/

/ (Division) 42
/ In Column 7 (Comment) 34

;

; (Punctuation) 37

>

>>CALL-CONVENTION 62
>>D 75
>>D (Debugging Line) 34
>>DEFINE 68
>>DISPLAY 76
>>ELIF 70
>>ELSE 70
>>END-IF 70
>>IF 69
>>LEAP-SECONDS 79
>>LISTING 78
>>PAGE 77
>>SET 71
>>SOURCE 73
>>TURN 74

^

^ (Exponentiation) 42

_

_ (Character in user-defined words) 10

0

01-Level Constants 146

6

66-Level Data Items 150

7

77-Level Data Items 151
78-Level Data Items 152

8

88-Level Data Items 153

A

A Sample Screen 23
ABS 426
ACCEPT 268
ACCEPT data-item 272
ACCEPT FROM COMMAND-LINE 269
ACCEPT FROM CONSOLE 268
ACCEPT FROM DATE/TIME 279
ACCEPT FROM ENVIRONMENT 271
ACCEPT FROM EXCEPTION STATUS 283
ACCEPT FROM Runtime-Info 281
ACCEPT FROM Screen-Info 280
ACCEPT OMITTED 282
ACCESS MODE DYNAMIC 113, 116
ACCESS MODE RANDOM 113, 116
ACCESS MODE SEQUENTIAL 109, 111, 116
ACOS 427
ADD 284
ADD CORRESPONDING 288
ADD GIVING 286
ADD TO 284
Additional Reference Sources 3
ADDRESS OF 384
ADVANCING PAGE 418
AFTER 361
AFTER ADVANCING 418
AFTER EXCEPTION CONDITION 254
ALL 101, 245, 414
ALL INTRINSIC 101
ALL OTHER 105
ALLOCATE 290
Alphabet-Name-Clause 96
Alphabetic Data Item 719
Alphabetic Data Items 195
ALPHABET 96
ALPHABETIC 339
Alphanumeric Data Item 719
Alphanumeric Data Items 195
Alphanumeric Literal 719
Alphanumeric Literal (Hexadecimal) 35
Alphanumeric Literal (Zero-Delimited) 35
Alphanumeric Literals 35
ALPHANUMERIC 339
ALPHANUMERIC-EDITED 339
ALSO 66, 321
Alternate Entry Points 681
ALTER 292
ALTERNATE RECORD KEY 116
An Example 611
ANNUITY 428
ANY 322
ANY LENGTH 154
Area 'A' (Columns 8-11) 30
Area 'B' (Columns 12-72) 30
ARGUMENT-NUMBER 269, 307
ARGUMENT-VALUE 270
Arithmetic Expressions 42
ASCENDING KEY 379
ASCII 96
ASIN 429
AT END 377
AT END + NOT AT END 256
ATAN 430
AUTHOR 85
AUTO 155
AUTO-SKIP 156
AUTOTERMINATE 157

B

BACKGROUND-COLOR 158
BASED 160
BEEP 161
BEFORE ADVANCING 418
BEFORE TIME 159
BELL 162
Binary Truncation 675
BIT-OF 431
BIT-TO-CHAR 432
BLANK 163
BLANK WHEN ZERO 164
BLINK 165
BLOCK CONTAINS 124
BOOLEAN-OF-INTEGER 533
Built-In System Subroutines 542
BY CONTENT 296, 686, 687
BY REFERENCE 248, 296, 686, 687
BY VALUE 248, 296, 687
BYTE-LENGTH 146, 433

C

C Main Programs Calling GnuCOBOL Subprograms ... 696
C$CALLEDBY ... 544
C$CHDIR ... 545
C$COPY ... 546
C$DELETE ... 547
C$FILEINFO ... 548
C$GETPID ... 549
C$JUSTIFY ... 550
C$MAKEDIR ... 551
C$NARG ... 552
C$PARAMSIZE ... 553
C$PRINTABLE ... 554
C$SLEEP ... 555
C$TOLOWER ... 556
C$TOUPPER ... 557
Call Environment ... 664
Called Program ... 679, 719
Called Program Considerations ... 687
Calling Program ... 679, 719
Calling Program Considerations ... 686
CALL ... 293
CALL-CONVENTION ... 93
CANCEL ... 298
Case Insensitivity ... 11
CBL_AND ... 558
CBL_CHANGE_DIR ... 559
CBL_CHECK_FILE_EXIST ... 560
CBL_CLOSE_FILE ... 561
CBL_COPY_FILE ... 562
CBL_CREATE_DIR ... 563
CBL_CREATE_FILE ... 564
CBL_DELETE_DIR ... 565
CBL_DELETE_FILE ... 566
CBL_EQ ... 567
CBL_ERROR_PROC ... 568
CBL_EXIT_PROC ... 570
CBL_FLUSH_FILE ... 572
CBL_GC_FORK ... 573
CBL_GC_GETOPT ... 575
CBL_GC_HOSTED ... 577
CBL_GC_NANOSLEEP ... 580
CBL_GC_PRINTABLE ... 581
CBL_GC_WAITPID ... 582
CBL_GET_CSR_POS ... 583
CBL_GET_CURRENT_DIR ... 584
CBL_GET_SCR_SIZE ... 585
CBL_IMP ... 586
CBL_NIMP ... 587
CBL_NOR ... 588
CBL_NOT ... 589
CBL_OPEN_FILE ... 590
CBL_OR ... 591
CBL_READ_FILE ... 592
CBL_READ_KBD_CHAR ... 593
CBL_RENAME_FILE ... 594
CBL_SET_CSR_POS ... 595
CBL_TOLOWER ... 596
CBL_TOUPPER ... 596
CBL_WRITE_FILE ... 597
CBL_XOR ... 598
CDF - Compiler Directing Facility ... 61
CHAR ... 434
CHAR-NATIONAL ... 534
Class Conditions ... 46
Class-Definition-Clause ... 98
CLASSIFICATION ... 91
CLOSE ... 299
COB-CRT-STATUS ... 93
COB_CC ... 645
COB_CFLAGS ... 645
COB_CONFIG_DIR ... 645
COB_CONFIG_DIR Environment Variable ... 649
COB_COPY_DIR ... 645
COB_DISPLAY_WARNINGS ... 658
COB_LDADD ... 645
COB_LDFLAGS ... 645
COB_LIBRARY_PATH ... 658
COB_LIBRARY_PATH Environment Variable ... 657, 658, 682
COB_LIBS ... 646
COB_LOAD_CASE ... 658
COB_LOAD_CASE Environment Variable ... 683
COB_PHYSICAL_CANCEL ... 658
COB_PHYSICAL_CANCEL Environment Variable ... 298, 683
COB_PRE_LOAD ... 659
COB_PRE_LOAD Environment Variable ... 295, 683
COB_SCREEN_ESC ... 659
COB_SCREEN_ESC Environment Variable ... 275
COB_SCREEN_EXCEPTIONS ... 659
COB_SCREEN_EXCEPTIONS Environment Variable ... 274, 275
COB_SET_DEBUG ... 659
COB_SET_DEBUG Environment Variable ... 255
COB_SET_TRACE ... 659
COB_SET_TRACE Environment Variable ... 370, 372
COB_SORT_MEMORY ... 659
COB_SORT_MEMORY Environment Variable ... 393
COB_SWITCH_n ... 659
COB_SWITCH_n Environment Variable ... 99
COB_SYNC ... 659
COB_TRACE_FILE ... 660
cobc - The GnuCOBOL Compiler - Runtime options ... 629
cobc option -Xref an example ... 641
COBCPY ... 646
COBCPY Environment Variable ... 648
cobcrun - Command-line Execution ... 656
COBOL Fundamentals ... 9

CODE-SET 125
Collating Sequence 719
COLLATING SEQUENCE 90, 105, 350
Color Palette 24
Color Palette and Video Attributes 23
Column 7 (Indicator Area) 30
Columns 1-6 (Sequence Number Area) 29
Columns 12-72 (Area B) 30
Columns 73-80 (Program Name Area) 30
Columns 8-11 (Area A) 30
COLUMN 166
COLUMNS 280
Combined Conditions 52
COMBINED-DATETIME 435
Combining GnuCOBOL and C Programs 692
COMMAND-LINE 269
Comments 33
COMMIT 300
Common Clauses on Executable Statements ... 256
COMMON 86
Compilation Group 640, 719
Compilation Time Environment Variables 645
Compilation Unit 640, 719
Compiler Configuration Files 649
Compiler Switches, -b 682
Compiler Switches, -conf 649
Compiler Switches, -debug 255, 449, 450, 537
Compiler Switches, -fdebugging-line 89, 255
Compiler Switches, -ffold-call 658
Compiler Switches, -ffold-copy 71, 648
Compiler Switches, -fintrinsics 421
Compiler Switches, -fintrinsics=ALL 101
Compiler Switches, -fixed 29, 71, 73
Compiler Switches, -foptional-file 105
Compiler Switches, -free 29, 71, 73
Compiler Switches, -fsyntax-extension 99
Compiler Switches, -ftrace 370, 372, 659
Compiler Switches, -ftraceall ... 372, 449, 450, 537, 659
Compiler Switches, -g 449, 450, 537
Compiler Switches, -I 648
Compiler Switches, -m 641, 656, 682
Compiler Switches, -o 641, 656
Compiler Switches, -O 678
Compiler Switches, -O2 678
Compiler Switches, -Os 678
Compiler Switches, -Wobsolete 86
Compiler Switches, -x 641, 656, 682, 721
Compiling Programs 629
COMPUTE 301
COMPUTE Versus ADD-SUBTRACT-MULTIPLY-DIVIDE 705
CONCAT 436
CONCATENATE 437
Concurrent Access to Files 57
Condition 710
Condition Names 45, 153
Conditional Expressions 45
CONFIGURATION SECTION 88
CONSOLE 95
CONSOLE IS CRT 93
CONSTANT 68, 71, 127, 168
Contained Subprograms 679
CONTENT 318
CONTENT-LENGTH 438
CONTENT-OF 439
Continuation (- in Column 7) 35
CONTINUE 303
Control Break 611, 719
Control Field 719
Control Footing 720
Control Heading 720
Control Hierarchy 609, 720
Control Hierarchy (Revisited) 624
CONTROL 137
CONVERSION 272, 309
CONVERTING 344
Copybook 14, 720
Copybook Naming Conventions and Usage 703
Copybooks 14
COPY 63
CORRESPONDING 258
COS 440
COUNT 414
Cross Reference listing using cobxref 643
CRT STATUS 93
CRT STATUS Codes 275
CURRENCY SIGN 93
CURRENCY-SYMBOL 441
current character pointer 401
CURRENT-DATE 442
CURSOR IS 93

D

D In Column 7 (Debugging Line) 34
Data 611, 711
Data Definition Principles 120
Data Description Clauses 154
Data Initialization 26
Data Item 14, 720
Data Item Coding and Naming Conventions ... 700
DATA DIVISION 119
DATA RECORD 124
Date Format 710
DATE-COMPILED 85
DATE-MODIFIED 85
DATE-OF-INTEGER 443
DATE-TO-YYYYMMDD 444
DATE-WRITTEN 85
DAY-OF-INTEGER 445
DAY-TO-YYYYDDD 446
DB_HOME 660
DB_HOME Environment Variable 59

DEBUG-ITEM Special Register 255
DEBUGGING MODE 89
DECIMAL POINT IS COMMA 93
DECLARATIVES 254
DEFAULT 340
DEFINED 70
DELETE 304
DELIMITED BY 402, 413
DELIMITER 414
DEPENDING ON 191
DESCENDING KEY 379
Detail Group 720
detail report 332
Detail Report 720
Direct Execution 656
DISPLAY 305
DISPLAY data-item 311
DISPLAY Data-Item 309
DISPLAY UPON COMMAND-LINE 307
DISPLAY UPON device 305
DISPLAY UPON ENVIRONMENT-NAME ... 308
DISPLAY-OF 535
DIVIDE 312
DIVIDE BY GIVING 316
DIVIDE INTO 312
DIVIDE INTO GIVING 314
Division 720
Divisions 32
Divisions Organize Programs 14
DUPLICATES 349, 391, 395
Dynamic Subprogram 681, 721
Dynamic vs Static Subprograms 681
Dynamically Loaded Subprograms 657

E

E 447
Elementary Item 14, 721
ELSE 338
EMPTY-CHECK 169
END-OF-PAGE 419
Entry-point 721
Entry-point Name 721
ENTRY 318
Environment Variables, COB_CONFIG_DIR .. 649
Environment Variables,
COB_LIBRARY_PATH 657, 658, 682
Environment Variables, COB_LOAD_CASE ... 683
Environment Variables,
COB_PHYSICAL_CANCEL 298, 683
Environment Variables,
COB_PRE_LOAD 295, 683
Environment Variables, COB_SCREEN_ESC .. 275
Environment Variables,
COB_SCREEN_EXCEPTIONS 274, 275
Environment Variables, COB_SET_DEBUG ... 255
Environment Variables,
COB_SET_TRACE 370, 372
Environment Variables,
COB_SORT_MEMORY 393
Environment Variables, COB_SWITCH_n 99
Environment Variables, COBCPY 648
Environment Variables, DB_HOME 59
Environment Variables, LANG 441, 492, 493, 494, 495
Environment Variables,
LD_LIBRARY_PATH 640
Environment Variables, PATH 682
Environment Variables, TEMP 126, 393
Environment Variables, TMP 126, 393
Environment Variables, TMPDIR 126, 393
Environment Variables: Compilation-Time 645
Environment Variables: Run-Time 658
ENVIRONMENT 271
ENVIRONMENT DIVISION 87
ENVIRONMENT-NAME 271
ENVIRONMENT-VALUE 271
EOL 170
EOS 170
ERASE 170
Error Exception Codes 451
Error Type Strings 451
ERROR 399
ESCAPE KEY 280
EVALUATE 320
EVENT STATUS 92
EXAMINE 324
EXCEPTION STATUS 281
EXCEPTION-FILE 448
EXCEPTION-FILE-N 536
EXCEPTION-LOCATION 449
EXCEPTION-LOCATION-N 537
EXCEPTION-STATEMENT 450
EXCEPTION-STATUS 451
Exceptions 761
Executable File 721
Executing Dynamically-Loadable Libraries 656
Execution Thread 721
EXHIBIT 326
EXIT 328
EXP 453
EXP10 454
EXTEND 359
EXTERNAL 171
EXTERNAL Data Items 688

F

FACTORIAL ... 455
FALSE ... 172
Figurative Constants ... 36, 721
File I/O ... 665
File I/O Environment Variables and/or dictionary file ... 673
File OPEN Modes ... 359
File Sharing ... 57
File Status Codes ... 107
File-Based SORT ... 391
File/Sort-Description ... 124
FILE SECTION ... 123
FILE STATUS ... 107
FILE-SECTION-Data-Item ... 127
Files ... 14
FILLER ... 121
FINAL ... 137
FIRST DETAIL ... 136
Fixed Format Mode ... 29, 73, 722
FOLDCOPYNAME ... 71
FOOTING ... 136
FOOTING AT ... 125
FOREGROUND-COLOR ... 173
FOREVER ... 360
Format of Program Source Lines ... 28
FORMATTED-CURRENT-DATE ... 456
FORMATTED-DATE ... 457
FORMATTED-DATETIME ... 458
FORMATTED-TIME ... 459
FRACTION-PART ... 460
Free Format Mode ... 29, 73, 722
FREE ... 331
FROM ... 174, 371, 374, 417
FROM CRT ... 272
FULL ... 175
FUNCTION ... 101
FUNCTION-ID ... 86
Functions ... 421

G

General Environment ... 661
General instructions ... 660
Generated Report Pages ... 618
GENERATE ... 332
GIVING ... 350, 393, 399, 405
GLOBAL ... 176
GLOBAL Data Items ... 687
Glossary of Terms ... 719
GNU Free Documentation License ... 767
GnuCobol use SQL for files ... 707
GnuCOBOL Compile time Options ... 708
GnuCOBOL Main Programs CALLing C Subprograms ... 694
GnuCOBOL Run-Time Library Requirements ... 692
GnuCOBOL Statements ... 268
GO TO ... 335
GO TO DEPENDING ON ... 336
GOBACK ... 334
Goto ... 710
Group Item ... 14, 722
GROUP INDICATE ... 177
Grouped Word Lists by feature and function ... 729

H

Header line ... 710
HEADING ... 136
HEX-OF ... 461
HEX-TO-CHAR ... 462
Hexadecimal Alphanumeric Literal ... 35, 722
Hexadecimal Numeric Literal ... 35, 722
HIGHEST-ALGEBRAIC ... 463
HIGHLIGHT ... 178
How RWCS Builds Report Pages ... 608

I

I-O ... 359
IDENTIFICATION DIVISION ... 85
Identifiers ... 722
IF ... 338
Imperative Statement ... 722
Independent Subprograms ... 679
Independent vs Contained vs Nested Subprograms ... 679
INDEXED BY ... 192, 379
Indexing ... 702
Indicator Area (Column 7) ... 30
INITIAL ... 86, 683
INITIALIZE ... 339
INITIALIZED ... 290
INITIATE ... 343
Inline PERFORM ... 362
INPUT ... 359
INPUT PROCEDURE ... 392
INPUT-OUTPUT SECTION ... 103
INSPECT ... 344
INSTALLATION ... 85
INTEGER ... 464
INTEGER-OF-BOOLEAN ... 538
INTEGER-OF-DATE ... 465
INTEGER-OF-DAY ... 466
INTEGER-OF-FORMATTED-DATE ... 467
INTEGER-PART ... 468
Interfacing to Other Environments ... 37
Interfacing With The OS ... 629
INTO ... 367, 369, 373
Intrinsic Function ... 723
Intrinsic Functions ... 421, 753
INTRINSIC ... 101
Introducing COBOL ... 4

Introduction 3
INVALID KEY + NOT INVALID KEY 259

J

JUSTIFIED 179

K

Key definition 712
KEY 192, 350, 369

L

Label 710
LABEL RECORD 124
LANG Environment Variable .. 441, 492, 493, 494, 495
Language Reserved Words 9
LAST CONTROL 136
LAST DETAIL 136
LD_LIBRARY_PATH 646
LD_LIBRARY_PATH Environment Variable .. 640
LEADING 64, 66, 213, 528
LEFTLINE 181
LENGTH 146, 469
LENGTH-AN 470
LENGTH-CHECK 182
Level Number 723
LINAGE IS n LINES 125
LINAGE-COUNTER Special Register 125, 419
LINE 183
LINE ADVANCING 15
LINE-COUNTER 215
LINE-COUNTER Special Register .. 138, 343, 409, 608
LINES 280
LINES AT BOTTOM 125
LINES AT TOP 125
LINKAGE SECTION 133
Literal 723
Literal Concatenation (&) 35
Literals 34
Literals (Alphanumeric) 35
Literals (Numeric) 34
LOCAL-STORAGE SECTION 131
LOCALE 93
LOCALE Names 93
LOCALE-COMPARE 471
LOCALE-DATE 472
LOCALE-TIME 473
LOCALE-TIME-FROM-SECONDS 474
Locating Copybooks 648
LOCK 108, 299
LOCK ON 59
LOG 475
LOG10 476
LOWER 71, 185
LOWER-CASE 477
LOWEST-ALGEBRAIC 478
LOWLIGHT 186

M

Main program 723
Main Program 679
Marking Changes in Programs 699
Matching C Data Types with GnuCOBOL USAGE's 693
MAX 479
MEAN 480
MEDIAN 481
MEMORY SIZE 90
MERGE 349
MIDRANGE 482
MIN 483
MOD 484
MODE IS BLOCK 272
MODULE-CALLER-ID 485
MODULE-DATE 486
MODULE-FORMATTED-DATE 487
MODULE-ID 488
MODULE-PATH 489
MODULE-SOURCE 490
MODULE-TIME 491
MONETARY-DECIMAL-POINT 492
MONETARY-THOUSANDS-SEPARATOR ... 493
MOVE 352
MOVE CORRESPONDING 353
multiple record locking 59
MULTIPLE FILE 118
MULTIPLY 354
MULTIPLY BY 354
MULTIPLY GIVING 356

N

National Character set 194, 723
NATIONAL 340
NATIONAL-EDITED 340
NATIONAL-OF 539
NATIVE 96
Negated Conditions 53
Nested Subprograms 680
NEXT 187, 366
NEXT GROUP 187
NEXT PAGE 183, 187
NEXT SENTENCE 338
NO ADVANCING 305
NO REWIND 299, 358
NO UPDATE 189
NO-ECHO 188
NOFOLDCOPYNAME 71
NORMAL 399

NOT 53
NOT INVALID KEY 260
NOT ON EXCEPTION 260
NOT ON OVERFLOW 261
NOT ON SIZE ERROR 261
Null Terminated Literals 35
Numeric Data Item 723
Numeric Data Items 195
Numeric Edited 196
Numeric Edited Data Item 723
Numeric Literal 724
Numeric Literal (Hexadecimal) 35
Numeric Literals 34
NUMERIC 339
NUMERIC SIGN TRAILING SEPARATE 95
NUMERIC-DECIMAL-POINT 494
NUMERIC-EDITED 340
NUMERIC-THOUSANDS-SEPARATOR 495
NUMVAL 496
NUMVAL-C 497
NUMVAL-C-V2 499
NUMVAL-F 500

O

OBJECT-COMPUTER 90
OCCURS 190
OFF 68
OFF STATUS 99
OMITTED 249
ON EXCEPTION + NOT ON EXCEPTION .. 260
ON OVERFLOW + NOT ON OVERFLOW ... 260
ON SIZE ERROR + NOT ON SIZE ERROR 261
ON STATUS 99
OPEN 358
OPTIONAL 105, 249
ORD 501
ORD-MAX 502
ORD-MIN 503
ORGANISATION INDEXED 18
ORGANISATION LINE SEQUENTIAL 15
ORGANISATION RELATIVE 17
ORGANISATION SEQUENTIAL 16
ORGANIZATION INDEXED 115
ORGANIZATION LINE SEQUENTIAL 111
ORGANIZATION RELATIVE 113
ORGANIZATION SEQUENTIAL 109
OUTPUT 359
OUTPUT PROCEDURE 350, 393
overflow condition 402
OVERLINE 193
OVERRIDE 68

P

Page Footing 724
Page Heading 724
PAGE 187
PAGE LIMITS 136
PAGE-COUNTER 215
PAGE-COUNTER Special Register .. 138, 343, 409
PARAMETER 68
PATH 660
PATH Environment Variable 682
perform scope 360
PERFORM 360
PI 504
PICTURE 142, 194
POINTER 401
Predefined Compilation Variables 646
Predefined compilation variables 82
PRESENT WHEN 201
PRESENT-VALUE 505
PREVIOUS 366
Primary Entry-Point 724
PRINTER 95
PRINTING 63
Procedural PERFORM 360
Procedure 724
Procedure name 724
Procedure Names 253
PROCEDURE DIVISION 247
PROCEDURE DIVISION CHAINING 250
PROCEDURE DIVISION RETURNING 252
PROCEDURE DIVISION Sections and Paragraphs 253
PROCEDURE DIVISION Sections Versus Paragraphs 704
PROCEDURE DIVISION USING 248
Procedures 253
Program 613, 724
Program Arguments 675
Program Name Area (Columns 73-80) 30
Program Structure 31
PROGRAM-ID 86
Programmer Productivity 6
Programming for XFD 707
Programming Style Suggestions 699
PROMPT 202
PROTECTED 203
Punctuation 37

Q

Qualification 724
Qualification of Data Names 39

R

Random READ ... 368
RANDOM ... 506
RANGE ... 508
Readability of Programs ... 11
READ ... 366
READY TRACE ... 370
Record ... 127, 724
Record Locking ... 59
RECORD CONTAINS ... 126
RECORD DELIMITER ... 105
RECORD IS VARYING ... 126
RECORD KEY ... 116, 304
RECORDING MODE ... 124
Recursive Subprogram ... 689
Recursive Subprograms ... 689
RECURSIVE ... 86
REDEFINES ... 204
REEL ... 299
Reference Modifiers ... 40
REFERENCE ... 318
Relation Conditions ... 50
RELATIVE KEY ... 113, 114, 304
RELEASE ... 371
REM ... 509
REMAINDER ... 314, 316
REMARKS ... 85
RENAMES ... 205
REPLACE ... 65
REPLACING ... 64, 65, 340, 344
Report Footing ... 724
Report Group ... 725
Report Group Definitions ... 139
Report Heading ... 725
Report I/O ... 673
Report Writer Features ... 25
Report Writer Usage ... 605
REPORT IS ... 126
REPORT SECTION ... 135
REPORT SECTION Data Items ... 141
REPOSITORY ... 101
REQUIRED ... 206
Reserved Word ... 725
Reserved Word Lists ... 727
Reserved Words ... 9, 729
RESERVE ... 105
RESET ... 216
RESET TRACE ... 372
RETURN ... 373
RETURN-CODE Special Register .. 294, 399, 543, 544, 545, 546, 547, 548, 549, 551, 553, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 570, 584, 586, 587, 588, 589, 590, 591, 592, 594, 597, 598, 679, 684, 685
RETURNING ... 290, 296, 399, 684
REVERSE ... 510
REVERSE-VIDEO ... 207
REVERSED ... 358
REWRITE ... 374
ROLLBACK ... 376
ROUNDED ... 261
Run Time Environment Variables ... 658
Running Programs ... 656
RUN ... 399
RWCS Lexicon ... 605

S

SAME AS ... 208
SAME RECORD AREA ... 117
SAME SORT ... 117
SAME SORT-MERGE ... 117
Screen Formatting Features ... 22
Screen I/O ... 670
SCREEN CONTROL ... 92
SCREEN SECTION ... 143
SCROLL ... 274
SCROLL DOWN ... 210
SCROLL UP ... 211
Search Index ... 377
SEARCH ... 377
SEARCH ALL ... 379
SECONDS-FROM-FORMATTED-TIME ... 511
SECONDS-PAST-MIDNIGHT ... 512
SECURE ... 212
SECURITY ... 85
SEGMENT-LIMIT ... 90
SELECT ... 104
Sentence ... 54, 725
SEPARATE CHARACTER ... 213
Sequence Number Area (Columns 1-6) ... 29
Sequential READ ... 366
SET ... 70, 382
SET ADDRESS ... 384
SET ATTRIBUTE ... 389
SET Condition Name ... 387
SET ENVIRONMENT ... 382
SET Index ... 385
SET LAST EXCEPTION ... 390
SET Program-Pointer ... 383
SET Switch ... 388
SET UP/DOWN ... 386
Sharing Data Between Calling and Called Programs ... 686
SHARING ... 108, 359
Sign Conditions ... 48
SIGN ... 513
SIGN IS ... 213
Simple GO TO ... 335
Simple MOVE ... 352
single record locking ... 59
SIN ... 514
SIZE ... 214
So What is GnuCOBOL? ... 8

SORT STATUS 107
Sorting and Merging Data 19
SORT 391
Source Line Format, Fixed 29, 73
Source Line Format, Free 29, 73
SOURCE 215
SOURCE IS 116
SOURCE-COMPUTER 89
Sparse Keys 116
Special Data Items 146
Special Registers 265, 725
Special Registers, DEBUG-ITEM 255
Special Registers, LINAGE-COUNTER .. 125, 419
Special Registers, LINE-COUNTER 138, 343, 409, 608
Special Registers, PAGE-COUNTER 138, 343, 409
Special Registers, RETURN-CODE 294, 399, 543, 544, 545, 546, 547, 548, 549, 551, 553, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 570, 584, 586, 587, 588, 589, 590, 591, 592, 594, 597, 598, 679, 684, 685
SPECIAL-NAMES 92
Split Keys 116
SQRT 515
STANDARD-1 96
STANDARD-2 96
STANDARD-COMPARE 540
STANDARD-DEVIATION 516
START 397
Statement 54, 725
Static Subprogram 681, 725
STATIC 294
STATUS 399
STDCALL 294
STDERR 95
STDIN 95
STDOUT 95
STEP 192
STOP 399
STORED-CHAR-LENGTH 517
String Allocation Differences Between GnuCOBOL and C 692
String Manipulation 20
STRING 401
Structured Data 14
Sub-Programming 679
Subprogram 679, 725
Subprogram Arguments 686
Subprogram Execution Flow 683
Subprogram Types 679
Subroutine 679, 725
Subroutine Execution Flow 683
Subscripting 702
SUBSTITUTE 518
SUBSTITUTE-CASE 519
SUBTRACT 403
SUBTRACT CORRESPONDING 407
SUBTRACT FROM 403
SUBTRACT GIVING 405
Summary of Document Changes 775
Summary Report 725
summary report 332
SUM 520
SUM OF 216
SUPPRESS 63, 409
SUPPRESS WHEN 116
Switch-Definition-Clause 99
Switch-Status Conditions 49
SWITCH-n 99
SWn 99
Symbolic-Characters-Clause 100
SYNCRONIZED 218
Syntax Diagram Conventions 27
SYSERR 95
SYSIN 95
SYSIPT 95
SYSLIST 95
SYSLST 95
SYSOUT 95
System names 759
System routines 757
SYSTEM 599

T

Table Handling 19
Table References 38
Table SORT 395
Table Subscripting versus Table Indexing 702
TALLYING 344
TAN 521
TEMP 660
TEMP Environment Variable 126, 393
TERMINATE 410
TEST-DATE-YYYYMMDD 522
TEST-DAY-YYYYDDD 523
TEST-FORMATTED-DATETIME 524
TEST-NUMVAL 525
TEST-NUMVAL-C 526
TEST-NUMVAL-F 527
The Anatomy of a Report 606
The Anatomy of a Report Page 607
The COBOL Language - Advanced Techniques 38
The COBOL Language - The Basics 9
THRU 321
TIME OUT 220
TIMEOUT 274
TIMES 360
TMP 646, 660
TMP Environment Variable 126, 393
TMPDIR 646, 660
TMPDIR Environment Variable 126, 393

TO 221
TRAILING 64, 66, 213, 528
TRANSFORM 411
TRIM 528
Turning PHYSICAL Page Formatting Into LOGICAL Formatting 627
TYPE 222
TYPEDEF 227

U

UNDERLINE 229
UNLOCK 412
UNSIGNED 241
UNSTRING 413
UNTIL 360
UNTIL EXIT 360
UPDATE 230, 274
UPON 216, 305
UPON CRT 309
UPON CRT-UNDER 309
UPPER 71, 231
UPPER-CASE 529
USAGE 232
Use of Periods 54
Use of VERB/END-VERB Constructs 55
USE AFTER STANDARD ERROR PROCEDURE 256
USE BEFORE REPORTING 254
USE FOR DEBUGGING 255
User-Defined Function 679, 725
User-Defined Function Execution Flow 684
User-Defined Names 725
User-Defined Words 10
USER NAME 281
USING 243, 248, 296, 318, 350, 392

V

VALUE 244, 318, 340
VALUE OF 124
VARIANCE 530
VARYING 192, 363, 377
Verb 55, 726

W

WHEN 321, 379
WHEN OTHER 321
WHEN-COMPILED 531
Why YOU Should Learn COBOL 4
WITH FILLER 340
WITH IGNORE LOCK 60
WITH LOCK 359
WITH TEST 361
WORKING-STORAGE SECTION 129
WRITE 417

X

X"91" 600
X"E4" 601
X"E5" 601
X"F4" 602
X"F5" 603
XFD Directives 707

Y

YEAR-TO-YYYY 532

Z

Zero-Delimited Alphanumeric Literals 35, 726