

opensource COBOL

Programmer's Guide

1st Edition, 31 July 2023

翻訳 OSS コンソーシアム
オープン COBOL ソリューション部会

Document Copyright © 2009, 2010 Gary Cutler
Document Copyright © 2021-2023 OSS コンソーシアム

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 no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the FDL is included in the section entitled “GNU Free Documentation License”.

改訂履歴

版		発行日
初版	2023 年 7 月 31 日	原文“OpenCOBOL 1.1 Programmer’s Guide”を参考 に日本語翻訳マニュアルを作成。
	2023 年 9 月 21 日	誤字や翻訳漏れを修正。
	2023 年 10 月 16 日	条件名の訳語を一部修正。

目次

1. まえがき	1
1.1. opensource COBOL とは.....	1
1.2. COBOL/opensource COBOL の重要機能	2
1.2.1. COBOL プログラムの構文.....	2
1.2.2. コピーブック	2
1.2.3. 構造化データ	2
1.2.4. ファイル	3
1.2.5. 表操作.....	8
1.2.6. データの並び替えと結合	8
1.2.7. 文字列操作	9
1.2.8. テキストユーザインターフェース(TUI)機能	12
1.3. 構文規則	13
1.4. ソースコードの形式.....	14
1.5. カンマ/セミコロンの使い方	16
1.6. COPY 文の使い方	17
1.7. 定数の使い方	18
1.7.1. 数字定数.....	18
1.7.2. 英数字定数.....	18
1.8. 表意定数の使い方.....	20
1.9. ユーザ定義名	21
1.10. LENGTH OF の使い方	21
2. opensource COBOL のプログラム形式.....	22
2.1. ネストされたユーザプログラム.....	24
2.2. ネストされたユーザ定義関数	25
3. 見出し部	26
4. 環境部	27
4.1. 構成節	28
4.1.1. 翻訳用計算機段落	28
4.1.2. 実行用計算機段落	29
4.1.3. リポジトリ段落.....	30
4.1.4. 特殊名段落	31
4.2. 入出力節	36

4.2.1. ファイル管理段落.....	37
4.2.1.1. 順編成ファイル.....	41
4.2.1.2. 相対編成ファイル.....	42
4.2.1.3. 索引編成ファイル.....	44
4.2.2. 入出力管理段落.....	46
5. データ部.....	47
5.1. ファイル記述.....	49
5.2. 整列用記述.....	52
5.3. データ記述の形式.....	54
5.4. 条件名.....	79
5.5. 定数記述.....	80
5.6. 画面記述.....	80
6. 手続き部.....	87
6.1. 構成要素.....	87
6.1.1. 表の参照.....	87
6.1.2. データ名の修飾.....	88
6.1.3. 部分参照.....	90
6.1.4. 式.....	91
6.1.4.1. 算術式.....	92
6.1.4.2. 条件式.....	96
6.1.5. ピリオド(.).....	102
6.1.6. 動詞 / END-動詞.....	104
6.1.7. 特殊レジスタ.....	106
6.1.8. ファイルへの同時アクセス制御.....	108
6.1.8.1. ファイル共有.....	108
6.1.8.2. レコードロック.....	109
6.2. 記述形式.....	108
6.3. 宣言の記述形式.....	113
6.4. ACCEPT.....	115
6.4.1. ACCEPT 文の書き方 1 — コンソールからの読み取り.....	115
6.4.2. ACCEPT 文の書き方 2 — コマンドライン引数の取得.....	115
6.4.3. ACCEPT 文の書き方 3 — 環境変数値の取得.....	117
6.4.4. ACCEPT 文の書き方 4 — 画面データの取得.....	118

opensource COBOL 1.1 Programmers Guide	目次
--	----

6.4.5. ACCEPT 文の書き方 5 — 日付/時刻の取得	120
6.4.6. ACCEPT 文の書き方 6 — 画面サイズデータの取得	122
6.4.7. ACCEPT 文の例外処理	122
6.5. ADD	123
6.5.1. ADD 文の書き方 1 — ADD TO	123
6.5.2. ADD 文の書き方 2 — ADD GIVING	125
6.5.3. ADD 文の書き方 3 — ADD CORRESPONDING	126
6.6. ALLOCATE	127
6.7. CALL	129
6.8. CANCEL	134
6.9. CLOSE	135
6.10. COMMIT	137
6.11. COMPUTE	138
6.12. CONTINUE	139
6.13. DELETE	141
6.14. DISPLAY	143
6.14.1. DISPLAY 文の書き方 1 — UPON CONSOLE	143
6.14.2. DISPLAY 文の書き方 2 — コマンドライン引数へのアクセス	144
6.14.3. DISPLAY 文の書き方 3 — 環境変数へのアクセスまたは設定	144
6.14.4. DISPLAY 文の書き方 4 — 画面データ	146
6.14.5. DISPLAY 文の例外処理	147
6.15. DIVIDE	148
6.15.1. DIVIDE 文の書き方 1 — DIVIDE INTO	148
6.15.2. DIVIDE 文の書き方 2 — DIVIDE INTO GIVING	149
6.15.3. DIVIDE 文の書き方 3 — DIVIDE BY GIVING	150
6.15.4. DIVIDE 文の書き方 4 — DIVIDE INTO REMAINDER	151
6.15.5. DIVIDE 文の書き方 5 — DIVIDE BY REMAINDER	152
6.16. ENTRY	153
6.17. EVALUATE	154
6.18. EXIT	158
6.19. FREE	161
6.20. GENERATE	162
6.21. GOBACK	163

6.22. GO TO	164
6.22.1. GO TO 文の書き方 1 — GO TO	164
6.22.2. GO TO 文の書き方 2 — GO TO DEPENDING ON	164
6.23. IF	166
6.24. INITIALIZE	167
6.25. INITIATE.....	169
6.26. INSPECT	170
6.27. MERGE	175
6.28. MOVE	178
6.28.1. MOVE 文の書き方 1 — MOVE	178
6.28.2. MOVE 文の書き方 2 — MOVE CORRESPONDING.....	178
6.29. MULTIPLY.....	182
6.29.1. MULTIPLY 文の書き方 1 — MULTIPLY BY	182
6.29.2. MULTIPLY 文の書き方 2 — MULTIPLY GIVING	183
6.30. NEXT SENTENCE	184
6.31. OPEN	185
6.32. PERFORM	187
6.32.1. PERFORM 文の書き方 1 — 手続き型	187
6.32.2. PERFORM 文の書き方 2 — インライン型	190
6.33. READ	191
6.33.1. READ 文の書き方 1 — 順次読み取り	191
6.33.2. READ 文の書き方 2 — ランダム読み取り	193
6.34. RELEASE	196
6.35. RETURN.....	197
6.36. REWRITE.....	198
6.37. ROLLBACK	200
6.38. SEARCH.....	201
6.38.1. SEARCH 文の書き方 1 — 順次探索	201
6.38.2. SEARCH 文の書き方 2 — 二分探索(SEARCH ALL)	203
6.39. SET.....	207
6.39.1. SET 文の書き方 1 — 環境設定	207
6.39.2. SET 文の書き方 2 — プログラムポインター設定	207
6.39.3. SET 文の書き方 3 — アドレス設定	208

opensource COBOL 1.1 Programmers Guide	目次
--	----

6.39.4. SET 文の書き方 4 — インデックス設定	209
6.39.5. SET 文の書き方 5 — UP / DOWN 設定	209
6.39.6. SET 文の書き方 6 — 条件名設定	210
6.39.7. SET 文の書き方 7 — スイッチ設定	211
6.40. SORT	212
6.40.1. SORT 文の書き方 1 — ファイルソート	212
6.40.2. SORT 文の書き方 2 — テーブルソート	217
6.41. START	219
6.42. STOP	222
6.43. STRING	223
6.44. SUBTRACT	225
6.44.1. SUBTRACT 文の書き方 1 — SUBTRACT FROM	225
6.44.2. SUBTRACT 文の書き方 2 — SUBTRACT GIVING	226
6.44.3. SUBTRACT 文の書き方 3 — SUBTRACT CORRESPONDING	227
6.45. SUPPRESS	228
6.46. TERMINATE	229
6.47. TRANSFORM	230
6.48. UNLOCK	232
6.49. UNSTRING	233
6.50. WRITE	237
7. opensource COBOL システムインターフェース	242
7.1. opensource COBOL コンパイラの使い方(cobc)	242
7.1.1. 解説	242
7.1.2. 構文とオプション	242
7.1.3. 実行可能プログラムのコンパイル	244
7.1.4. 動的にロード可能なサブプログラム	244
7.1.5. 静的サブルーチン	246
7.1.6. COBOL と C プログラムの結合	247
7.1.6.1. opensource COBOL ランタイムライブラリの要件	247
7.1.6.2. opensource COBOL と C の文字列割り当ての違い	247
7.1.6.3. C データ型と opensource COBOL USAGE 句の一致	249
7.1.6.4. opensource COBOL メインプログラムの C サブプログラム呼び出し	251
7.1.6.5. C メインプログラムの opensource COBOL サブプログラム呼び出し	253

opensource COBOL 1.1 Programmers Guide	目次
--	----

7.1.7. 重要な環境変数.....	255
7.1.8. コンパイル時のコピーブックの検索.....	257
7.1.9. コンパイラ構成ファイルの使い方	258
7.2. opensource COBOL プログラムの実行.....	260
7.2.1. プログラムの直接実行.....	260
7.2.2. 「cobcrun」ユーティリティの使用	261
7.2.3. プログラムの引数.....	262
7.2.4. 重要な環境変数.....	263
7.3. 組み込みサブルーチン	265
7.3.1. 「名前による呼び出し」ルーチン	265
7.3.1.1. CALL “C\$CHDIR” USING <i>directory-path, result</i>	266
7.3.1.2. CALL “C\$COPY” USING <i>src-file-path, dest-file-path, 0</i>	267
7.3.1.3. CALL “C\$DELETE” USING <i>file-path, 0</i>	267
7.3.1.4. CALL “C\$FILEINFO” USING <i>file-path, file-info</i>	267
7.3.1.5. CALL “C\$JUSTIFY” USING <i>data-item, “justification-type”</i>	268
7.3.1.6. CALL “C\$MAKEDIR” USING <i>dir-path</i>	268
7.3.1.7. CALL “C\$NARG” USING <i>arg-count-result</i>	269
7.3.1.8. CALL “C\$PARAMSIZE” USING <i>argument-number</i>	269
7.3.1.9. CALL “C\$SLEEP” USING <i>seconds-to-sleep</i>	269
7.3.1.10. CALL “C\$TOLOWER” USING <i>data-item, BY VALUE convert-length</i>	270
7.3.1.11. CALL “C\$TOUPPER” USING <i>data-item, BY VALUE convert-length</i>	270
7.3.1.12. CALL “CBL_AND” USING <i>item-1, item-2, BY VALUE byte-length</i>	270
7.3.1.13. CALL “CBL_CHANGE_DIR” USING <i>directory-path</i>	271
7.3.1.14. CALL “CBL_CHECK_FILE_EXIST” USING <i>file-path, file-info</i>	271
7.3.1.15. CALL “CBL_CHANGE_DIR” USING <i>directory-path</i>	272
7.3.1.16. CALL “CBL_COPY_FILE” USING <i>src-file-path, dest-file-path</i>	273
7.3.1.17. CALL “CBL_CREATE_DIR” USING <i>dir-path</i>	273
7.3.1.18. CALL “CBL_CREATE_FILE” USING <i>file-path, 2, 0, 0, file-handle</i>	273
7.3.1.19. CALL “CBL_DELETE_DIR” USING <i>dir-path</i>	274
7.3.1.20. CALL “CBL_DELETE_FILE” USING <i>file-path</i>	274
7.3.1.21. CALL “CBL_ERROR_PROC” USING <i>function, program-pointer</i>	275
7.3.1.22. CALL “CBL_EXIT_PROC” USING <i>function, program-pointer</i>	277
7.3.1.23. CALL “CBL_EQ” USING <i>item-1, item-2, BY VALUE byte-length</i>	279

7.3.1.24.	CALL “CBL_FLUSH_FILE” USING <i>file-handle</i>	279
7.3.1.25.	CALL “CBL_GET_CURRENT_DIR” USING BY VALUE 0, BY VALUE <i>length</i> , BY REFERENCE <i>buffer</i>	279
7.3.1.26.	CALL “CBL_IMP” USING <i>item-1</i> , <i>item-2</i> , BY VALUE <i>byte-length</i>	280
7.3.1.27.	CALL “CBL_NIMP” USING <i>item-1</i> , <i>item-2</i> , BY VALUE <i>byte-length</i>	281
7.3.1.28.	CALL “CBL_NOR” USING <i>item-1</i> , <i>item-2</i> , BY VALUE <i>byte-length</i>	281
7.3.1.29.	CALL “CBL_NOT” USING <i>item-1</i> , BY VALUE <i>byte-length</i>	282
7.3.1.30.	CALL “CBL_OC_NANOSLEEP” USING <i>nanoseconds-to-sleep</i>	282
7.3.1.31.	CALL “CBL_OPEN_FILE” <i>file-path</i> , <i>access-mode</i> , 0, 0, <i>handle</i>	283
7.3.1.32.	CALL “CBL_OR” USING <i>item-1</i> , <i>item-2</i> , BY VALUE <i>byte-length</i>	284
7.3.1.33.	CALL “CBL_READ_FILE” USING <i>handle</i> , <i>offset</i> , <i>nbytes</i> , <i>flag</i> , <i>buffer</i>	284
7.3.1.34.	CALL “CBL_RENAME_FILE” USING <i>old-file-path</i> , <i>new-file-path</i>	285
7.3.1.35.	CALL “CBL_TOLOWER” USING <i>data-item</i> , BY VALUE <i>convert-length</i>	285
7.3.1.36.	CALL “CBL_Toupper” USING <i>data-item</i> , BY VALUE <i>convert-length</i>	286
7.3.1.37.	CALL “CBL_WRITE_FILE” USING <i>handle</i> , <i>offset</i> , <i>nbytes</i> , 0, <i>buffer</i>	286
7.3.1.38.	CALL “CBL_XOR” USING <i>item-1</i> , <i>item-2</i> , BY VALUE <i>byte-length</i>	287
7.3.1.39.	CALL “SYSTEM” USING <i>command</i>	287
8.	サンプルプログラム	289
8.1.	FileStat-Msgs.cpy – ファイル状態コード	289
8.2.	COBDUMP -16 進数/文字データダンプサブルーチン	290

1. まえがき

1.1. opensource COBOL とは

このマニュアルでは、opensource COBOL の最新版に実装されているプログラミング言語 COBOL の構文、意味、利用法について紹介する。

opensource COBOL とは OSS コンソーシアムで開発・公開している COBOL コンパイラであり、2012 年に OpenCOBOL(開発者 Keisuke Nishida さんと Roger While さん)からフォークし、PIC N(2 バイト文字)を代表とする日本語拡張や国産汎用機の互換性機能など、日本の商習慣に応じて独自機能を追加したプロダクトである。

opensource COBOL は COBOL を C 言語にトランスレートし、gcc などの C コンパイラでバイナリを生成する。

Linux 用として開発されたが、Mac OS や、Linux 互換の仮想環境である Cygwin や MinGW¹を利用することで、Windows でも構築可能である。また C コンパイラや、リンカー/ローダーを提供する Microsoft の Visual Studio を利用することで、ネイティブ Windows アプリケーションとして構築できる。

¹ MinGW はたった一つの DLL で opensource COBOL コンパイラやランタイムを作成して、opensource COBOL のツールとユーザプログラムが利用できる。DLL は GNU 一般公衆利用承諾書(General Public License)の定める条件下であれば無償で配布が可能である。MinGW によって構築された opensource COBOL は、128MB のフラッシュドライブに簡単に適合して実行でき、利用時に Windows にソフトウェアをインストールする必要もない。ただし、同時に実行している opensource COBOL プログラム間でのファイル共有処理や、特定のファイル型の レコードロック処理など、一部の言語機能は利用できない。

1.2. COBOL/opensource COBOL の重要機能

1.2.1. COBOL プログラムの構文

COBOL プログラムは、部(DIVISION)として知られる、それぞれ独自の目的を持つ 4 つの主要なコーディング領域で構成されている。

部は様々な節(SECTION)で構成され、節は 1 つ以上の段落(PARAGRAPH)で構成される。更に段落は完結文(SENTENCE)で構成され、完結文は 1 つ以上の文(STATEMENT)で構成される。

このプログラム構成要素の階層構造により、すべての COBOL プログラムの構成が標準化される。このマニュアルの大部分は、COBOL プログラムを構成する様々な部、節、段落、および文について説明している。

4 つの部とその機能については 2 章で、各部についてはそれぞれの章(3、4、5、および 6 章)で説明する。

1.2.2. コピーブック

「コピーブック」とは、プログラムに COPY 文(1.6)を使用してそのコードをインポートするだけで、複数のプログラムで利用できるプログラムコードの部品であり、ファイル、データ構造、または手続き型コードを定義できる。

現在のプログラミング言語には、これと同じ機能を実行する文(通常は「include」または「#include」)がある。ただし、COBOL コピーブック機能が現在の言語の「include」機能と異なるのは、COBOL の COPY 文はインポートされたソースコードをコピーしながら編集できるということである。この機能により、コピーブックライブラリはコードの再利用することができる。

1.2.3. 構造化データ

COBOL は 1960 年代に構造化データの概念を導入した。構造化データは、単一の項目とし

てアクセスできるデータ、または構造内の文字の出現位置に基づいて従属項目に分割できるデータである。これらの構造は**集団項目**と呼ばれる。構造の一番下には、従属項目に分割されていないデータ項目がある。COBOL では、これらを**基本項目**と呼ぶ。

1.2.4. ファイル

COBOL の主な強みの一つは、様々なファイルにアクセスできることである。opensource COBOL は、他の COBOL 実装と同様に、読み書きするファイルの構造を記述しておく必要がある。ファイル構造の最高レベルの特性は、次のように、ファイルの編成(4.2.1)を指定することによって定義される。

ORGANIZATION IS 内部構造の中で最も単純なファイルであり、その内容は一連のデータレコードとして簡単に構造化され、特殊なレコード終了区切り文字で終了する。ASCII 改行文字 (16 進数の 0A) は、UNIX または疑似 UNIX (MinGW、Cygwin、MacOS) の opensource COBOL ビルドで使用されるレコード終了区切り文字である。真のネイティブ Windows ビルドでは、行頭復帰(CR)、改行(LF) (16 進数の 0D0A) 順序が使用される。

LINE SEQUENTIAL

ファイルタイプのレコードは、同じ長さである必要はない。

レコードは、純粹にファイルの先頭から順に読み書きする必要がある。レコード番号 100 を読み取る(または書き込む) 唯一の方法は、最初にレコード番号 1 から 99 を読み取る(または書き込む) ことである。

opensource COBOL プログラムによってファイルに書き込まれるとき、区切り文字順序が各データレコードに自動的に追加される。

ファイルが読み取られるとき、opensource COBOL ランタイムシステムは各レコードから末尾の区切り文字順序を削除し、読み取ったデータがプログラム内のデータレコード用に記述された領域よりも短い場合、必要に応じて、データ(の右側)を空白で埋める。データが長すぎる場合は切り捨てられ、超過分は消失する。

これらのファイルは、正確なバイナリデータ項目を含むように定義してはならない。これらの項目の内容の値の一部として、誤ってレコード終了順序が含まれる可能性があるためである。これは、ファイル読み取り時にランタイムシステムを混乱させ、その値を実際のレコード終了順序として解釈してしまう。

ORGANIZATION IS
RECORD BINARY
SEQUENTIAL

これらのファイルも単純な内部構造を持っており、内容も一連の固定長データレコードとして簡単に構化されており、特別なレコード終了区切り文字はない。

このファイルタイプのレコードは、物理的な長さがすべて同じである。可変長論理レコードがプログラムに定義されている場合(5.3)、ファイル内の各物理レコードが占有する空白は、占有可能な最大である。

レコードは、純粹にファイルの先頭から順に読み書きする必要がある。レコード番号 100 を読み取る(または書き込む)唯一の方法は、最初にレコード番号 1 から 99 を読み取る(または書き込む)ことである。

ファイルが opensource COBOL プログラムによって書き込まれる場合、区切り文字順序はデータに追加されない。

ファイルが読み取られると、データはファイルに存在する通りにプログラムに転送される。短いレコードが最後のレコードとして読み取られる場合は空白が埋め込まれる。

このようなファイルを読み取るプログラムは、そのファイルを作成したプログラムが使用する長さと同じ長さのレコードを記述するよう注意しなければならない。例えば、次の例は 6 文字のレコードを 5 つ書き込んだプログラムによって作成された RECORD BINARY SEQUENTIAL ファイルの内容を示している。「A」、「B」、… の値と背景色は、ファイルに書き込まれたレコードを反映している。

ここで、別のプログラムがこのファイルを読み取るが、6 文字ではなく 10 文字のレコードが記述されているとする。プログラムが読み取るレコードは次の通りである。

これはあなたが求めている結果かもしれないが、多くの場合でこれは望ましい動作ではない。これは、コピーブックを使用してファイルのレコードレイアウトを記述することで、そのファイルにアクセスする複数のプログラムが同じレコードサイズとレイアウトを「参照する」ことが保証される。

これらのファイルには、正確なバイナリデータ項目を含めることができる。レコード終了区切り文字がないため、レコード項目の内容は読み取りプロセスとは無関係である。

ORGANIZATION IS RELATIVE

ファイルの内容は、4 バイトの USAGE COMP-5(表 5-10)レコードヘッダーで始まる一連の固定長データレコードで構成される。レコードヘッダーにはデータの長さがバイト単位で含まれるが、バイト数には 4 バイトのレコードヘッダーは含まれない。

このファイルタイプのレコードは、物理的な長さがすべて同じである。可変長論理レコードがプログラムに定義されている場合(5.3)、ファイル内の各物理レコードが占有する空白は、占有可能な最大である。

このファイル構成は、順次処理またはランダム処理に対応するように定義されている。相対ファイルを使用すると、最初にレコード 1 から 99 を読み書きする必要はなく、レコード 100 を直接読み書きできる。opensource COBOL ランタイムシステムは、プログラムで定義された最大レコードサイズを使用して、レコードヘッダーとデータが開始するファイル内の相対バイト位置を計算し、必要なデ

ータをプログラムとの間で転送する。

ファイルが opensource COBOL プログラムによって書き込まれる場合、区切り文字順序はデータに追加されないが、各物理レコードの先頭にレコード長項目が追加される。

ファイルが読み取られると、データはファイルに存在する通りにプログラムに転送される。

このようなファイルを読み取るプログラムは、そのファイルを作成したプログラムが使用する長さと同じ長さのレコードを記述するよう注意しなければならない。ファイルからプログラムにデータを転送するときに、opensource COBOL ランタイムライブラリが4バイトのASCII文字列をレコード長として解釈してしまうと、問題となる場合がある。

これは、コピーブックを使用してファイルのレコードレイアウトを記述することで、そのファイルにアクセスする複数のプログラムが同じレコードサイズとレイアウトを「参照する」ことが保証される。

これらのファイルには、正確なバイナリデータ項目を含めることができる。レコード終了区切り文字がないため、レコード項目の内容は読み取りプロセスとは無関係である。

ORGANIZATION IS INDEXED

opensource COBOL プログラムで利用できる最も高度なファイル構造である。使用する opensource COBOL ビルドに含まれている高度なファイル管理機能(Berkeley DB[BDB]、VBISAM など)によって構造が異なるため、ファイルの物理構造を説明することはできない。代わりに、ファイルの論理構造について説明する。

索引ファイルには複数の構造が格納される。一つ目は、相対ファイルの内部構造に似ていると考えられるデータ構成要素である。ただし、データレコードは相対ファイルのように、レコード番号で直接アクセスすることも、ファイル内の物理的な順序で順次処理することもできない。

残りの構造は、1 つ以上の索引構成要素となり、これは(どうにかして) 各データレコード内の主キーと呼ばれる項目内容(お客様番号、従業員番号、商品コード、氏名等)をレコード番号に変換するデータ構造である。これにより、特定の主キー値のデータレコードを直接読み取り、書き込み、削除することができる。更に、索引データ構造は、主キー項目値の昇順でファイルをレコードごとに順次処理できるように定義されている。構造の動作については説明した通りで、この索引構造がバイナリ検索可能なツリー構造 (btree) として存在するか、精巧なハッシュ構造であるかどうか、プログラマには関係ない。ランタイムシステムは、同じ主キー値を持つ 2 つのレコードを索引付きファイルに書き込むことを許可しない。

追加項目を代替キーとして定義する機能がある。一つの例外を除いて、代替キー項目は主キーと同じように動作し、代替キー項目値に基づいてレコードデータへの直接アクセスと順次アクセスの両方を許可する。その例外とは、代替キー項目が opensource COBOL コンパイラにどのように記述されるかによって、代替キーが重複する値を持つことができる可能性があるということである (4.2.1.3)。

代替キーの数に制限はないが、各キー項目にはディスク容量と実行時間の制限が伴う。代替キー項目の数が増えると、ファイル内のレコードの書き込みや修正にかかる時間が更に長くなる。

これらのファイルには、正確なバイナリデータ項目を含めることができる。レコード終了区切り文字がないため、レコード項目の内容は読み取りプロセスとは無関係である。

すべてのファイルは、環境部の入出力節のファイル管理段落でコーディングされた SELECT 文(4.2.1) を使用して、最初に opensource COBOL プログラムに記述される。SELECT 文では、プログラム内で参照されるファイル名を定義することに加えて、ファイル編成、ロック(6.1.9.2)と共有(6.1.9.1)オプションも一緒に、オペレーティングシステムに認識される名前とパスを指定する。

データ部の作業場所節のファイル節にあるファイル記述(5.1)は、可変長レコードが可能か

どうか—可能な場合—最小長と最大長はどのくらいか、ということを含むファイル内のレコードの構造を定義する。更に、ファイル記述項は、ファイル入出力のブロックサイズを指定できる。

1.2.5. 表操作

他のプログラミング言語にある配列と基本的に同じものとして、COBOL には表がある。COBOL の表機能を特別なものにしているのは、COBOL 言語に存在する 2 つの文—SEARCH (6.38.1) と SEARCH ALL (6.38.2) である。

1 つ目は表を順次検索し、任意の数の検索条件のうち 1 つに一致する表記述項が見つかった場合、またはすべての表記述項が検索され、いずれの条件にも一致しない場合にのみ停止する。

2 つ目は、それぞれの表記述項に含まれる「キー」項目で並び替えおよび検索された表に対して、非常に高速に検索を実行できる。このような検索に使用されるアルゴリズムは、バイナリ検索（半区間検索とも呼ばれる）と言い、目的の記述項を見つけるため、または目的の記述項が表に存在しないことを確認するために、表の少数の記述項のみを検索する必要があることが保証される。表が大きいほど、この検索方法はより効果的である。例えば、32,768 の記述項がある表でも特定の記述項を見つけることができ、15 記述項以下の検索で記述項が存在しないと判断することができる。このアルゴリズムは、SEARCH ALL(6.38.2)で詳しく説明している。

1.2.6. データの並び替えと結合

COBOL 言語には、任意の複雑なキー構造に従って大量のデータを並び替えることができる強力な SORT 文(6.40.1)がある。このデータは、プログラム内で生成される場合もあれば、1 つ以上の外部ファイルのものを扱う場合もある。並び替えられたデータは、1 つ以上の出力ファイルに自動的に書き込まれるか、並び替えられた順番でレコードごとに処理される。

表のデータを並び替えるためだけの特別な形式の SORT 文 (6.40.2) も存在し、表に対して SEARCH ALL を使用する場合に特に便利である。

opensource COBOL Programmers Guide	まえがき
------------------------------------	------

同類の文 —MERGE (6.27)—では、複数のファイルの内容を結合できるが、ファイルはすべて同じキー構造に従って同様の方法で並べ替えられる。出力結果は、入力ファイルの内容で構成されており、結合されると共通のキー構造に従って順序付けられ、1 つ以上の出力ファイルに自動的に書き込まれるか、プログラムによって内部的に処理される。

1.2.7. 文字列操作

テキスト文字列の処理専用設計されたプログラミング言語があり、強力な数値計算を実行することのみを目的として設計されたプログラミング言語があります。ほとんどのプログラミング言語は、これら 2 つの両極端の間に位置します。COBOL も例外ではありませんが、非常に強力な文字列操作機能が含まれています。実際、opensource COBOL には、他の多くの COBOL 実装よりもさらに多くの文字列操作機能があります。次の表は、文字列に関する opensource COBOL の機能を示しています。

機能	サポートする opensource COBOL 機能
2 つ以上の文字列を連結する	CONCATENATE 組み込み関数(6.1.7.9) STRING 文(6.43)
数値型で定義されている時刻または日付を書式文字列に変換する	LOCALE-TIME(6.1.7.31) または LOCALE-DATE 組み込み関数(6.1.7.30)
バイナリ値をプログラムの文字セットに対応する文字に変換する	CHAR 組み込み関数 (6.1.7.7) 関数を呼び出す前に引数に 1 を追加する。 CHAR 関数の説明では、数値型引数の値に 1 を追加しなくても同じ結果が得られる MOVE 文の利用法を示している。
文字列を小文字に変換する	LOWER-CASE 組み込み関数(6.1.7.35) C\$TOLOWER 組み込みサブルーチン (7.3.1.10) CBL_TOLOWER 組み込みサブルーチン (7.3.1.35)

opensource COBOL Programmers Guide	まえがき
------------------------------------	------

文字列を大文字に変換する	UPPER-CASE 組み込み関数(6.1.7.67) C\$TOUPPER 組み込みサブルーチン(7.3.1.11) CBL_TOUPPER 組み込みサブルーチン(7.3.1.36)
文字をプログラムの文字セットに対応する数値に変換する	ORD 組み込み関数 (6.1.7) 結果から 1 を引く。ORD 関数の説明では、数値型引数の値に 1 を追加しなくても同じ結果が得られる MOVE 文の利用法を示している。
文字列内にある部分文字列の出現回数をカウントする	TALLYING オプションを指定した INSPECT 文(6.26)
数値書式指定文字列を復号して数値に戻す(例えば「\$12,342.19-」を「-12342.19」という値に復号する)。	NUMVAL 組み込み関数(6.1.7.42) NUMVAL-C 組み込み関数(6.1.7.43)
文字列または文字列を格納できるデータ項目の長さを決定する	LENGTH 組み込み関数(6.1.7.29) または BYTE-LENGTH 組み込み関数 (6.1.7.6)
文字列の開始位置と長さに基づいて部分文字列を抽出する	「送信」項目に部分参照を含む MOVE 文(6.28.1)
桁区切り記号 (アメリカでは「,」)、通貨記号 (アメリカでは「\$」)、小数点、クレジット/デビット記号、先頭または末尾の記号文字を含む、出力用の数値項目を書式化する。	受け取り項目に適用された PICTURE 編集記号(5.3)を指定した MOVE 文(6.28)
文字列項目の位置揃え (左、右、または中央)	C\$JUSTIFY 組み込みサブルーチン(7.3.1.5)
文字列内の 1 つ以上の文字を異なる文字で単アルファベット置換する	CONVERTING オプションを指定した INSPECT 文(6.26) TRANSFORM 文(6.47) SUBSTITUT 組み込み関数(6.1.7.60)

opensource COBOL Programmers Guide	まえがき
------------------------------------	------

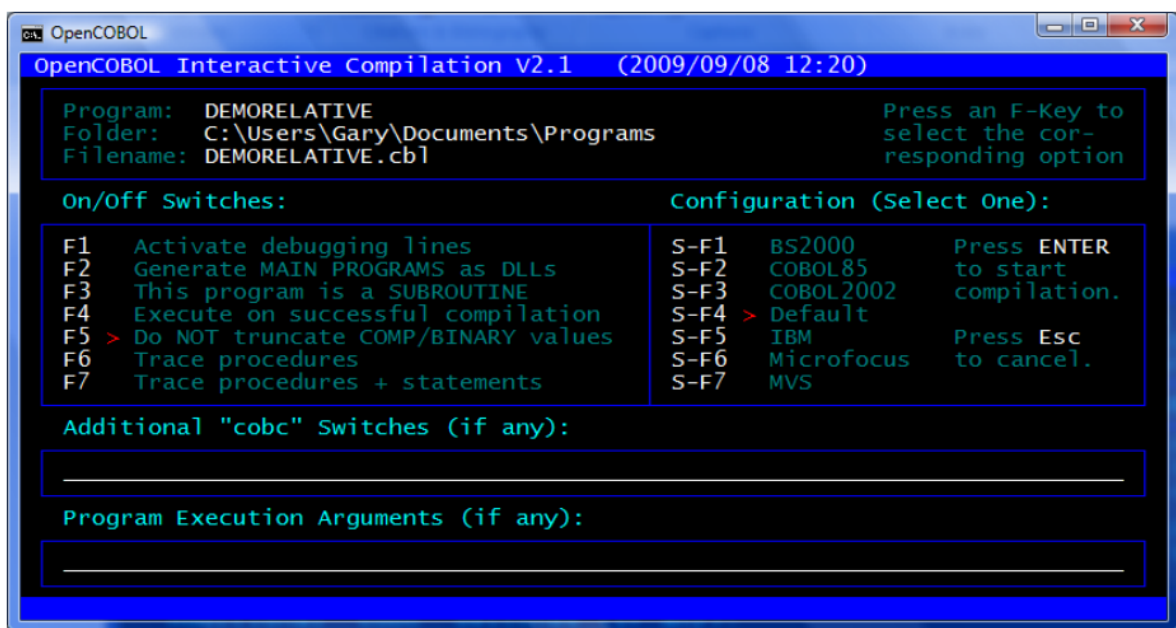
	および SUBSTITUTE-CASE 組 み 込 み 関 数 (6.1.7.61)
文字列を解析し、1 つ以上の区切り文字順序に基づいて部分文字列に分割する これらの区切り文字は、単一の文字、複数の文字列、またはいずれかが重複した連続オカレンスの可能性がある	UNSTRING 文(6.49)
文字列の先頭または末尾の空白の削除	TRIM 組み込み関数 (6.1.7.66)
部分文字列の開始文字位置と長さに基づいて、単一の部分文字列を <u>同じ長さの別の部分文字列</u> に置換する	「受け取り」項目に部分参照を含む MOVE 文 (6.28.1)
文字列内にある 1 つ以上の部分文字列を、オカレンス位置に関係なく、 <u>同じ長さの置換部分文字列</u> に置換する	REPLACING オ プ シ ョ ン を 指 定 し た INSPECT 文 (6.26) SUBSTITUTE 組み込み関数(6.1.7.60) および SUBSTITUTE-CASE 組 み 込 み 関 数 (6.1.7.61)
文字列内にある 1 つ以上の部分文字列を、オカレンス位置に関係なく、 <u>異なる長さの置換部分文字列</u> に置換する	SUBSTITUTE 組み込み関数(6.1.7.60) および SUBSTITUTE-CASE 組 み 込 み 関 数 (6.1.7.61)

1.2.8. テキストユーザインターフェース(TUI)機能

COBOL2002 標準は、テキストベースの画面の定義と処理を可能にする COBOL 言語の拡張機能を形式化している。opensource COBOL は、COBOL2002 で説明されている画面処理機能を実質的にすべて実装している。

以下は、Windows コンピュータのコンソールウィンドウに表示される画面の例である。

図 1-1-TUI サンプル画面



このような画面²は、データ部(5.6)の画面節で定義され、一度定義されると、画面はACCEPT文(6.4.4)およびDISPLAY文(6.14.4)を介して実行時に再度使用される。

COBOL2002 標準は、テキストユーザインターフェース(TUI)画面のみを対象としており、最新のオペレーティングシステムに組み込まれている、より高度なグラフィカルユーザインターフェース(GUI)画面設計および処理機能は対象ではない。完全な GUI 開発ができるサブルーチンベースのパッケージが利用可能ではあるが、どれもオープンソースではない。

² この画面は、OCic という名前のプログラム—opensource COBOL コンパイラのフルスクリーンフロントエンド—のものである。

1.3. 構文規則

opensource COBOL 言語の構文について、COBOL プログラマに馴染みのある規則に従って説明していく。以下は、構文の記述方法についての説明である。

大文字 COBOL 言語のキーワードと実装に依存する名前(いわゆる「予約語」)は大文字で表示される。

下線 下線が引かれている予約語は、構文上の文脈により必要である。予約語に下線が引かれていない場合はオプションであり、プログラムに影響を与えない。

小文字 置換可能な引数を表す一般的な用語は小文字で表示される。

[] 角括弧は、オプションの句を囲むために使われ、囲まれていない句は必須である。

| 単純な選択は、縦線で区切って示される場合がある。COBOL 構文図では通常使われないが、角括弧によって構文図が複雑になりすぎる場合に効果的な代替手段である。

{ } 中括弧は、選択肢を囲むために使われ、選択肢の中から一つを正確に選択する必要がある。

{ | }

選択指示子は、囲まれた選択肢の中から一つ以上が選択される可能性がある選択肢を囲むために使われる。

… 角括弧、中括弧、セレクトー、または小文字記述項の後に表示される 3 つの点(「省略記号」と呼ばれる)は、省略記号の前の構文要素が複数回出現する可能性があることを示す。

網掛け部分 網掛け部分は、opensource COBOL コンパイラによって認識されるが、

生成されたコードに影響を与えないか、サポートされていないものとして拒否される構文要素を強調するために使われる。このような要素は、他の COBOL 環境からのプログラム移行を容易にするために opensource COBOL 言語に存在するか、まだ完全に実装されていない、または廃止された構文要素を反映する。

1.4. ソースコードの形式

従来の COBOL プログラムソースコードは、固定形式の 80 文字(最大)行を使用してコーディングしていたが、ANSI 2002 規格では自由形式が定義されており、ソースコードの長さは最大 256 文字で、特定桁に固定の意味の割り当てはない。

opensource COBOL には、入力ファイルのソースコード形式を指定する、次の四つの方法がある。

-fixed

この opensource COBOL コンパイラスイッチは、ソースコード入力が従来の固定形式(80 桁)になることを指定し、これが初期モードである。

-free

この opensource COBOL コンパイラスイッチは、ソースコード入力 ANSI2002 の自由形式(256 桁)になることを指定する。

>>SOURCE FORMAT IS FREE

このソース行は、opensource COBOL コンパイラが検出すると、コンパイラは自由書式を受け付ける。「>>」文字は、8 桁目以降で開始する必要がある。これと次の命令を使用することで、コンパイラを自由モードと固定モード間で自由に切り替えることができる。

>>SOURCE FORMAT IS FIXED

このソース行は、opensource COBOL コンパイラが検出すると、コンパイラは固定書式を受け付ける。これと前の命令を使用することで、コンパイラを自由モードと固定モード間で自由に切り替えることができる。

以下のものは、opensource COBOL プログラムで様々なことを示すために使う、特別な命令または文字である。

7 桁目の「*」

ソース行がコメントであることを示し、固定形式モードの場合のみ有効である。

7 桁目の「D」

ソース行が有効な opensource COBOL コードであり、opensource COBOL コンパイラに「-fdebugging-line」スイッチが指定されていない限り(その場合、行はコンパイルされる)コメントであることを示す。固定形式モードの場合のみ有効である。

任意の桁の「*>」

ソース行の残りの部分がコメントであることを示す。自由形式モードと固定形式モードのどちらでも使用できるが、固定形式モードで使用する場合は、「*」を 7 桁目以降に入力する必要がある。

任意の桁の「>>D」

ソース行が有効な opensource COBOL コードであり、opensource COBOL コンパイラに「-fdebugging-line」スイッチが指定されていない限り(その場合、行はコンパイルされる)コメントであることを示す。固定形式モードと自由形式モードのどちらの場合でも有効である。自由形式モードではどの桁からでも開始できるが、固定形式モードでは、8 桁目以降から開始しなければならない。

1.5. カンマ/セミコロンの使い方

空白が有効な場所(もちろん英数字定数内を除く)での読みやすさ向上のために、コンマ文字(,)またはセミicolon(;)を opensource COBOL プログラムにオプションとして挿入できる。COBOL 標準ではコンマを使用する場合、コンマの後に少なくとも一つの空白を続ける必要がある。最近、COBOL コンパイラ(opensource COBOL を含む)の多くは、この規則を緩和して、ほとんどの場合で空白を省略できるようになったが、これにより、DECIMAL POINT IS COMMA 句が使用されている場合(4.1.4 を参照)、コンパイラに「混乱」が生じる可能性がある。

次の文では、二つの引数(数字定数 1 および 2)を渡すサブルーチンを呼び出す：

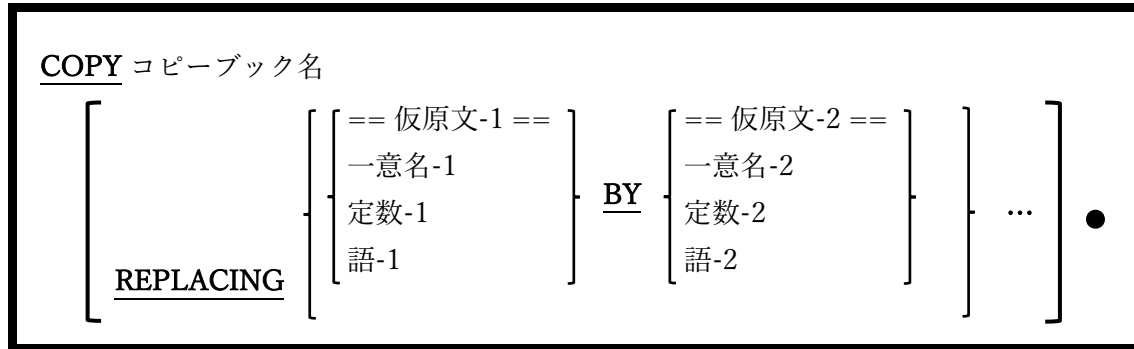
```
CALL  "SUBROUTINE"  USING  1,2
```

DECIMAL POINT IS COMMA を使用すると、実際には一つの引数(非整数データ型数字定数 1 および 2)を呼び出すサブルーチンとして解釈される。

句読点としてのコンマの後に空白をコーディングする習慣を付けたい。別の方法としては、「混乱」の可能性をなくするためにセミicolonのコーディングが考えられる。

1.6. COPY 文の使い方

図 1-2-COPY 構文



COPY 文は、プログラムにコピーブック(1.2.2)をインポートするために使われる。

opensource COBOL は、コピーブックの使用を完全にサポートしている。コピーブックとは、COPY 文も入れた全ての COBOL 構文を含む個別のソースファイルのことである。

COPY 文は、コピーブックに含まれるコードが構文的に有効である COBOL プログラム内のどこでも使用できる。

上記の構文図では、COPY 文の最後のピリオドと、REPLACING 句を強調している。経験のある COBOL プログラマの目には、ピリオドがあるべきではないと思われる場合でも、全ての COPY 文の最後にはピリオドが絶対に必須である。

コンパイルプロセスが開始される前に、全ての COPY 文が決定され、対応するコピーブックの内容がプログラムのソースコードに挿入される。

オプションの REPLACING 句を使用すると、予約語(語-1、語-2)、データ項目(一意名-1、一意名-2)、定数(定数-1、定数-2)、または空白で区切られたフレーズを置き換えることができる。コピーブックがプログラムに含まれているため、何度でも置換を行うことができる。

プログラムのコンパイル時に opensource COBOL コンパイラがコピーブックを見つける方法の詳細については、「7.1.8 コンパイル時のコピーブックの検索」で説明する。

1.7. 定数の使い方

定数は定数値であり、プログラムの実行中に変更されることはない。定数には、数値と英数値の二つの基本型がある。

1.7.1. 数字定数

数字定数は、配列の添え字として、算術式の値として、または数値の使用可能な手続き型文で使うことができる数字定数であり、次のいずれかの形式をとる。

- 1、56、2192、-54 などの整数。
- 1.12 や-2.95 などの整数でない固定小数点値。
- H' 1F'(1F₁₆ = 31₁₀)、h'22'(22₁₆ = 34₁₀)、H'DEAD'(DEAD₁₆ = 57005₁₀)などの 16 進数定数。「H」文字は大文字または小文字のいずれかであり、一重引用符(')または二重引用符(")のいずれかの文字を使用できる。16 進数定数は、H'FFFFFFFFFFFFFFFF'(64 ビット値)を最大値として制限されている。

1.7.2. 英数字定数

英数字定数は、コンピュータ画面での表示、レポートへの印刷、通信接続を介した伝送、または PIC X または PIC A データ項目への格納に適した文字列である(5.3)。これらは、同等の数値計算に変換できない限り、算術式で使用することはできない(6.1.7 の NUMVAL および NUMVAL-C 組み込み関数を参照)。

英数字定数は、次の形式のいずれかを取ることができる。

- 一重引用符(')文字または二重引用符(")文字で囲まれた一連の文字は、**文字列定数**を構成する。二重引用符(")は定数内のデータ文字として使用することができる。データ文字として一重引用符文字を含める必要がある場合は、一重引用符を 2 つ続けて('')表現することで、一重引用符(')を定数内のデータ文字として使用することができる。二重

引用符をデータ文字として含める必要がある場合は、二重引用符を 2 つ続けて(" ")表現する。

- X"4A4B4C"(4A4B4C₁₆ = ASCII 文字列「JKL」)、x'20'(20₁₆ = 空白)、X'30313233'(30313233₁₆ = ASCII 文字列「0123」)などの 16 進数定数。「X」文字は大文字または小文字のいずれかで、一重引用符(')または二重引用符(")文字を使用できる。16 進数の英数字定数は、各文字が 8 ビット分のデータ(2 桁の 16 進数)で表されるため、常に偶数の 16 進数で構成する必要がある。16 進英数字定数の長さはほぼ無制限である。

英数字定数が長すぎて 1 行に収まらない場合は、次の 2 つの方法のいずれかで次の行に続けることができる。

- ソースコード形式の固定モード(1.5)を使用している場合、英数字定数は 72 桁目まで実行できる。定数は、一重引用符または二重引用符(最初の行の定数を開始するときを使用した方)をコーディングすることにより、次の行の 11 桁目以降に続けることができる。次の行では 7 桁目にハイフン(-)をコーディングする必要がある。以下がその例である。

```

      1      2      3      4      5      6      7      8
1234567890123456789012345678901234567890123456789012345678901234567890
01  LONG-LITERAL-VALUE-DEMO      PIC X(60) VALUE "This is a long l
-                                     "iteral that must
-                                     " be continued."
```

- 現在のソースコード形式に関係なく、opensource COBOL では英数字定数を個別の断片に分割でき、それぞれに開始と終了の一重引用符または二重引用符があり、「&」文字を使用して「結合」されているため、7 桁目にハイフン(-)をコーディングする必要はない。以下がその例である。

```

      1      2      3      4      5      6      7      8
1234567890123456789012345678901234567890123456789012345678901234567890
01  LONG-LITERAL-VALUE-DEMO      PIC X(60) VALUE "This is a" &
                                     " long literal that must " &
                                     "be continued."
```

プログラムで自由モードのソースコード形式を使用している場合、文は 255 字にも及ぶ可

opensource COBOL Programmers Guide	まえがき
------------------------------------	------

能性があるため、長い英数字定数を続ける必要はほとんどない。

数字定数と予約語は、英数字定数と同じように、上記の方法のいずれかを使用して(予約語は1つ目の方法を使用して)複数の行に分割できるが、プログラムの見栄えが悪くなるため、この二つが分割されることは稀である。

1.8. 表意定数の使い方

表意定数は、特定の定数の代用となる予約語である。一般に、表意定数は対応する値が使用可能な場所であればどこでも自由に使用することができ、値の前に「ALL」が付いているかのように解釈される(「ALL」については5.3で説明する)。

次の表は、opensource COBOL の表意定数とそれぞれに対応する値を示している。

表 1-3-表意定数

表意定数	定数型	値
ZERO, ZEROS, ZEROES	数字	0
SPACE, SPACES	英数字	空白
QUOTE, QUOTES	英数字	二重引用符
LOW-VALUE, LOW-VALUES	英数字	プログラムの大小順序で値が最も小さい文字。プログラムが ASCII 大小順序を使用している場合、0 ビットで構成される一連の文字を表す。
HIGH-VALUE, HIGH-VALUES	英数字	プログラムの大小順序で値が最も大きい文字。プログラムが ASCII 大小順序を使用している場合、1 ビットで構成される一連の文字を表す。
NULL	英数字	ゼロビットで構成される文字 (プログラムの大小順序と無関係)。

1.9. ユーザ定義名

opensource COBOL プログラムを作成するときは、プログラムのあらゆる側面、プログラムデータ、およびプログラムが実行されている外部環境を表す様々な名称を定義する必要がある。

ユーザ定義名は、文字「A」から「Z」（大文字または小文字）、「0」から「9」、ダッシュ（「-」）およびアンダースコア（「_」）で構成され、ハイフンまたはアンダースコア文字で開始または終了することはできない。

プロシージャ名を除いて、ユーザ定義名には少なくとも 1 文字が含まれていなければならない。ユーザ定義名がデータの名称として作成される場合、このドキュメントでは一意名の下で参照される。

1.10. LENGTH OF の使い方

オプションで、英数字定数と一意名の前に「LENGTH OF」という句を付けることができる。この場合、実際の定数は、英数字定数のバイト数と等しい値を持つ数字定数である。例えば、次の二つの opensource COBOL 文はどちらも同じ結果(27)を表示する。

```
01 Demo-Identifier PIC X(27). *> This is a 27-character data-item
.
.
.
DISPLAY LENGTH OF "This is a LENGTH OF Example"
DISPLAY LENGTH OF Demo-Identifier
DISPLAY 27
```

定数または一意名参照の LENGTH OF 句は、通常、数値定数を指定できる場所であればどこでも使用できるが、次のように使用する場合は例外となる。

1. DISPLAY 文の定数の代わりとして
2. WRITE 文または RELEASE 文の FROM 句の一部として
3. PERFORM 文の TIMES 句の一部として

2. opensource COBOL のプログラム形式

図 2-1-opensource COBOL のプログラム形式

```
{[ IDENTIFICATION DIVISION . ]  
  PROGRAM-ID. プログラム名-1 [ IS INITIAL PROGRAM ] .  
  [ ENVIRONMENT DIVISION. 環境部記述]  
  [ DATA DIVISION. データ部記述 ]  
  [ PROCEDURE DIVISION. 手続き部記述 ]  
  [ ネストされたユーザ定義プログラム | ネストされたユーザ定義関数 ] ...  
  [ END PROGRAM プログラム名-1 . ] } ...
```

COBOL プログラムは、共通の目的に関連する言語文が主要なグループごとに分けられ、区分として編成されている。

すべてのプログラムにおいて区分けが必要なわけではないが、使用時に示されている順序で指定する必要がある。

1. opensource COBOL コンパイラは、ソースコード(コンパイルユニット)を単一の実行可能プログラムにコンパイルします。このソースコードは、単一のプログラム(プログラムに必要な区分によって定義され、後ろにオプションの END PROGRAM 句が続くソースコード順序)、または必須の区分と END PROGRAM 句で構成される複数のプログラムである。複数のプログラムが単一のコンパイルユニットでコンパイルされている場合、最後のプログラムに END PROGRAM 句を含める必要はないが、それ以外のプログラムには一つは必要である。
2. opensource COBOL コンパイラに複数の入力ファイルを指定すると、指定ファイルの内容で構成されたコンパイルユニットが定義され、指定された順序でコンパイルされる。効果は、複数のプログラムを含む単一のソースファイルがコンパイルされた場合と同じであるが、複数のプログラムが含まれていない限り、個々のソースファイルに END PROGRAM 句を含める必要はない。

opensource COBOL Programmers Guide	opensource COBOL のプログラム形式
------------------------------------	---------------------------

3. 単一のコンパイルユニットを構成するプログラムの数に関係なく、単一の出力実行可能プログラムのみ生成される。コンパイルユニットで最初に検出されたプログラムがメインプログラムとして機能し、それ以外のプログラムは、メインプログラムまたは他のプログラムによって順番に呼び出されるサブプログラムとして機能する。
4. 各区分の目的の概要は次の通りである：

区分	目的
見出し	プログラム ID(プログラム名)を指定することにより、プログラムの基本認証を定義する(3 章)。
環境	プログラムが動作する外部計算機環境を定義する区域で、プログラムがアクセスする可能性のあるファイルの定義を含む(4 章)。
データ	プログラムが処理するすべてのデータを定義する(5 章)。
手続き	すべての実行可能プログラムコードを含む(6 章)。

2.1. ネストされたユーザプログラム

図 2-2-ネストされたユーザプログラム

```
[ IDENTIFICATION DIVISION . ]  
  
  PROGRAM-ID. プログラム名-1 [ IS { INITIAL  
                                COMMON } PROGRAM ] .  
  
[ ENVIRONMENT DIVISION. 環境部記述 ]  
[ DATA DIVISION. データ部記述 ]  
[ PROCEDURE DIVISION. 手続き部記述 ]  
[ ネストされたユーザ定義プログラム | ネストされたユーザ定義関数 ] ...  
[ END PROGRAM プログラム名-1 . ]
```

ネストされたユーザプログラムは、他のプログラム内に埋め込まれたプログラムである(これらは「親」プログラムの手続き区分に従い、間に介在する END PROGRAM は存在しない)。そのため、埋め込まれている親プログラムでのみ使用可能なサブプログラムとして機能する³。

1. ネストされたユーザプログラム自体に、他のネストされたプログラムが含まれている場合がある。ネスト構造が「等しいレベル」であると考えられるネストされたサブプログラムの間に END PROGRAM 句を含めるよう注意しなければならない。

³ もちろん、すべてのルールには常に例外が存在する。26 ページ PROGRAM-ID 段落の COMMON 句で説明する。

2.2. ネストされたユーザ定義関数

図 2-3-ネストされたユーザ定義関数

```
FUNCTION-ID. 関数名-1 [ IS { INITIAL | COMMON } PROGRAM ].  
[ ENVIRONMENT DIVISION. 環境部記述 ]  
DATA DIVISION. データ部記述  
PROCEDURE DIVISION.  
    [ USING データ項目-1 ... ]  
    [ RETURNING データ項目-n ].  
    手続き-部記述  
[ ネストされたユーザ定義プログラム | ネストされたユーザ定義関数 ] ...  
[ END FUNCTION 関数名-1. ]
```

ユーザ定義関数は opensource COBOL の構文として定義されているが、現在はサポートされていない。

1. ユーザ定義関数をコンパイルしようとする、以下のようなメッセージが表示され、拒否される。

```
name:line: Error: FUNCTION-ID is not yet implemented
```

3. 見出し部

図 3-1-見出し部構文

[IDENTIFICATION DIVISION .]

PROGRAM-ID. プログラム名-1 [IS { INITIAL | COMMON } PROGRAM] .

プログラム ID(プログラム名)を指定することにより、プログラムの基本認証を定義する。

1. 見出し部 (IDENTIFICATION DIVISION) のヘッダーはオプションであるが、PROGRAM-ID 句はオプションではない。
2. PROGRAM-ID 句は他のプログラムが参照できるように(つまり CALL “program-name”)、名前(プログラム名)を定義する。
3. プログラム名は大文字と小文字を区別する。コンパイル単位が動的にロード可能なライブラリファイル(opensource COBOL コンパイラコマンドの「-m」オプションを使用するもの)として作成されている場合、コンパイラによって作成されたライブラリファイル名はプログラム名と完全に一致する。コンパイル単位が実行可能ファイル(opensource COBOL コンパイラコマンドの「-x」オプションを使用するもの)として作成されている場合、プログラム ID は有効な COBOL 一意名となり、実行可能ファイル名は、「cbl」または「cob」拡張子のないソースプログラムファイル名と同じになる。
4. INITIAL 句と COMMON 句は、サブプログラム内で使用される。COMMON 句はネストされたユーザプログラムであるサブプログラム内でのみ使うことができる。
5. INITIAL 句を指定すると、サブプログラムは最初だけでなく実行される度に、初期(つまりコンパイル済み)状態が確保される。
6. COMMON 句が存在している場合は、ネストされたユーザプログラム(サブプログラ

ム)ユニットを、親プログラムだけでなく、その親に当たる他のネストされたユーザプログラムでも使用できるようにする。

7. 「-Wobsolete」コンパイルスイッチが使用されていない限り、DATE-WRITTEN、DATE-COMPILED、AUTHOR、INSTALLATION、SECURITY、REMARKS などの廃止された見出し部記述項は、通常は無視される。このような場合、警告メッセージが生成されるがコンパイルは続行される。

4. 環境部

図 4-1-環境部構文

```
ENVIROMENT DIVISION.  
[ CONFIGRATION SECTION. ]  
[ INPUT-OUTPUT SECTION. ]
```

プログラムが動作する外部計算機環境を定義する区域で、プログラムがアクセスする可能性のあるファイルの定義を含む。

1. 環境部(ENVIRONMENT DIVISION)によって定義できる機能のいずれもプログラムで必要としない場合は、この区域を指定する必要はない。

4.1. 構成節

図 4-2-構成節構文

```
CONFIGURATION SECTION.  
[ SOURCE-COMPUTER. 翻訳用計算機記述 ]  
[ OBJECT-COMPUTER. 実行用計算機記述 ]  
[ REPOSITORY. リポジトリ記述 ]  
[ SPECIAL-NAMES. 特殊名記述 ]
```

プログラムがコンパイルおよび実行される計算機システムを定義し、特殊な環境構成や互換性特性も指定する。

1. 構成節(CONFIGURATION DIVISION)の段落が指定される順序に関連性はない。

4.1.1. 翻訳用計算機段落

図 4-3-翻訳用計算機段落構文

```
SOURCE-COMPUTER. 計算機名-1  
[ WITH DEBUGGING MODE ] .
```

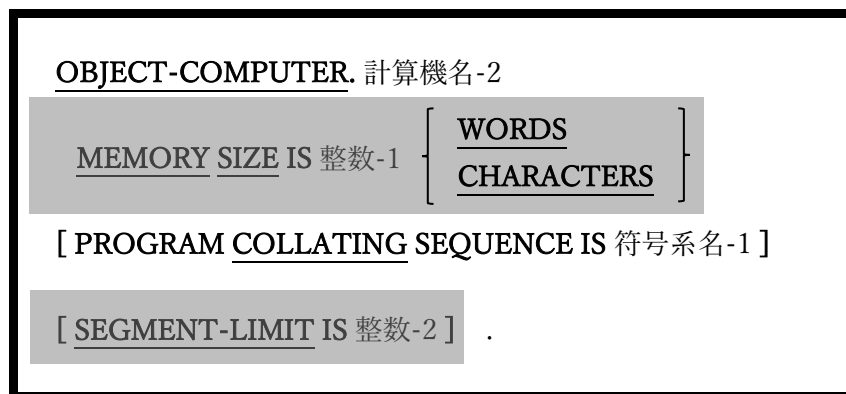
翻訳計算機(SOURCE-COMPUTER)段落は、プログラムがコンパイルされる計算機を定義する。

1. 計算機名-1 に指定された値が、opensource COBOL の予約語とは一致しない有効な COBOL 語である場合、この値は定義と無関係である。
2. オプションの WITH DEBUGGING MODE 句が存在する場合、廃止した構文としてフラグが付けられ(「-W」、「-Wobsolete」、または「-Wall」コンパイラスイッチを使う場合)、プログラムのコンパイルには影響しない。

- ただし、opensource COBOL コンパイラへの「**-fdebugging-line**」スイッチを指定することで、プログラムのデバッグ行をコンパイルできる。opensource COBOL プログラムでデバッグ行を指定する方法については 1.5 で説明している。

4.1.2. 実行用計算機段落

図 4-4-実行用計算機段落構文



実行用計算機(OBJECT-COMPUTER)段落は、プログラムが実行される計算機について説明する段落ではあるが、単なるドキュメントではない。

- 計算機名-2 に指定された値が、opensource COBOL の予約語とは一致しない有効な COBOL 語である場合、この値は定義と無関係である。
- MEMORY SIZE 句と SEGMENT-LIMIT 句は互換性の目的でサポートされているが、opensource COBOL では機能しない。
- PROGRAM COLLATING SEQUENCE 句を使用すると、英数字の値を相互に比較するときに用いる、カスタマイズされた文字の大小順序を指定できる。データは引き続き計算機に固有の文字セットに格納されるが、比較のために文字が並べ替えられる論理的な順序を計算機に固有の文字セットに変更できる。符号系名-1 は、特殊名節 (4.1.4) で定義する必要がある。

4. PROGRAM COLLATING SEQUENCE 句が指定されていない場合、計算機に固有の文字セット(通常は ASCII)によって暗示される大小順序が使用される。

4.1.3. リポジトリ段落

図 4-5-リポジトリ段落構文

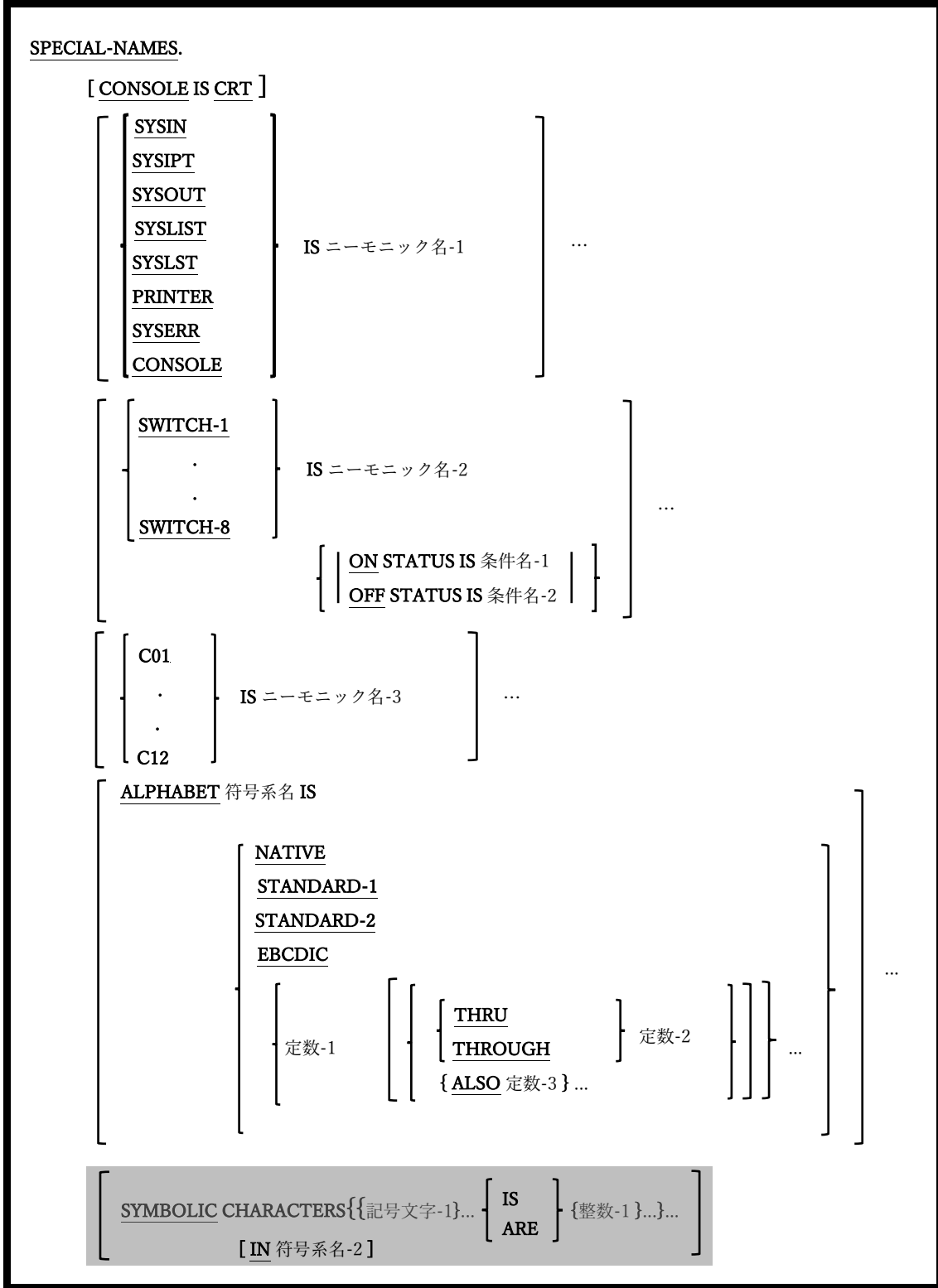
<u>REPOSITORY.</u> <u>FUNCTION</u> { 関数名-1... <u>ALL</u> } <u>INTRINSIC</u> .
--

リポジトリ(REPOSITORY)段落は、様々な組み込み関数へのアクセスを制御するためのメカニズムを定義する。

1. 関数名の前に「FUNCTION」とコーディングしなくても、一つ以上(またはすべて)の組み込み関数に使用可能とするフラグを立てることができる。組み込み関数については 6.1.7 で説明する。
2. この段落を使用する代わりに、「**-ffunctions-all**」スイッチを用いて opensource COBOL プログラムをコンパイルすることもできる。

4.1.4. 特殊名段落

図 4-6-特殊名段落構文




```

[ CLASS 字類名-1 IS 定数-4 { [ [ THRU
                           THROUGH ] 定数-5 ] ... } ... ] ...
[ LOCALE 局所名-1 IS 一意名-1 ] ...
[ CURRENCY SIGN IS 定数-6 ]
[ DECIMAL-POINT IS COMMA ]
[ CURSOR IS 一意名-2 ]
[ CRT STATUS IS 一意名-3 ]
[ SCREEN CONTROL IS 一意名-4 ]
[ EVENT STATUS IS 一意名-5 ] .

```

特殊名(SPECIAL-NAMES)段落は、通貨記号の指定、小数点の選択、[記号文字の指定]実装者名とユーザ指定のニーモニック名の関連付け、アルファベット名と文字セットまたは大小順序の関連付け、および字類名と文字のセットの関連付けを行う。

つまり、この段落には、別の PC 環境で作成された COBOL プログラムを簡単に「構成」して、opensource COBOL 環境では最小限の変更のみでコンパイルできるようにするといった役割がある。

1. `CONSOLE IS CRT` 句は、opensource COBOL の他のバージョンとのソースコードの互換性を保持する。これにより、デバイス「CRT」と「CONSOLE」を `DISPLAY` 文(6.14.1)および `ACCEPT` 文(6.4.1)で相互に使用できるようになる。opensource COBOL プログラムを「ゼロから」コーディングする場合は、これら二つのデバイスはすでに同様のものと見なされているため、この句は必要ない。
2. `IS` ニーモニック名-1 句を使うと、「IS」の前に指定された組み込み opensource COBOL デバイス名に代替名を定義することができる。
3. `SWITCH-1` から `SWITCH-8` の外部値は、それぞれ `COB_SWITCH_1` から `COB_SWITCH_8` の環境変数を使用してプログラムに指定される。「ON」の値はスイッチをオンにし、その他の値(未定義の環境変数を含む)はスイッチをオフにする。`ON STATUS` 句および `OFF STATUS` 句は、実行時にスイッチが設定されているかどうか

うかをテストするための条件名を定義する。詳細については 6.1.4.2.1 および 6.1.4.2.4 で説明する。

4. ALPHABET 句は、「定数-1」オプションを使用して自分で定義したものを含め、名前を、指定された文字コードセットまたは大小順序と関連付けることができ、定数-1、定数-2、または定数-3 に英数字定数を指定できる。比喩的な定数 SPACE [S]、ZERO [[E] S]、QUOTE [S]、HIGH-VALUE [S]、または LOW-VALUE [S] を指定することもできる。
5. SYMBOLIC CHARACTERS 句は構文的に認識されても無視される。「-Wall」または「-W」コンパイラスイッチを使用すると、この機能がまだ実装されていないことを示す警告メッセージが表示される。
6. ユーザ定義クラスは、CLASS 句を使って定義される。この句で指定された定数はクラスの一部と見なされるため、データ項目の値に含まれる可能性のある文字を定義する。例えば、以下に「Hexadecimal」と呼ばれるクラスを定義し、データ項目が「Hexadecimal」クラスの一部である場合、データ項目に存在する可能性のある文字のみを指定する。

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

このユーザ定義クラスの使用例については、6.1.4.2.2 で説明する。

LOCALE 句を使って、UNIX 標準のローカル名をデータ部で定義された一意名と関連付けることができ、局所名は次のいずれかになる：

表 4-7-局所名

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

7. CURRENCY SIGN 句を使って、PICTURE 編集記号で使用する通貨記号として任意の 1 文字を定義できる(表 5-9 を参照)。通貨記号が指定されていない場合の既定値はドル記号(\$)である。
8. DECIMAL POINT IS COMMA 句は、PICTURE 編集記号(表 5-9 を参照)および数字定数として使用される場合「,」および「.」文字の定義を逆にするが、望ましくない副作用が生じる可能性がある(1.6 を参照)。

9. 一意名-3 の PICTURE 句(CRT-STATUS)は 9(4)である必要がある。この項目は ACCEPT 画面の実行時ステータスを示す 4 桁の値を受け取り、ステータスコードは次の通りである。

表 4-8-ACCEPT 画面ステータスコード

コード	意味
0000	ENTER キー押下
1001 - 1064	F1 - F64
2001,2002	PgUP,PgDn ⁴
2003,2004,2006	上矢印,下矢印,PrtSc (プリントスクリーン) ⁵
2005	Esc ⁶
8000	ACCEPT 画面に利用できるデータがない
9000	致命的な I/O 画面エラー

10. CRT STATUS 句が指定されていない場合、ACCEPT ステータス画面を受け取る目的で、COB-CRT-STATUS 一意名(9(4)の PICTURE 句)が暗黙的に割り当てられる。
11. SCREEN CONTROL 句と EVENT STATUS 句は、コンパイル時にサポートされていない一方で、CURSORIS 句はサポートされている。しかし現在、実行時には機能していない。

⁴ 実行時に環境変数 COB_SCREEN_EXCEPTIONS が空白以外の値に設定されている場合にのみ使用できる。

⁵ Windows システムでは検出できない。

⁶ 実行時に環境変数 COB_SCREEN_ESC が空白以外の値に設定されている場合にのみ使用できる。(これは COB_SCREEN_EXCEPTIONS の設定に追加される)。

4.2. 入出力節

図 4-9-入出力節構文

INPUT-OUTPUT SECTION.

[FILE-CONTROL. ファイル管理記述]

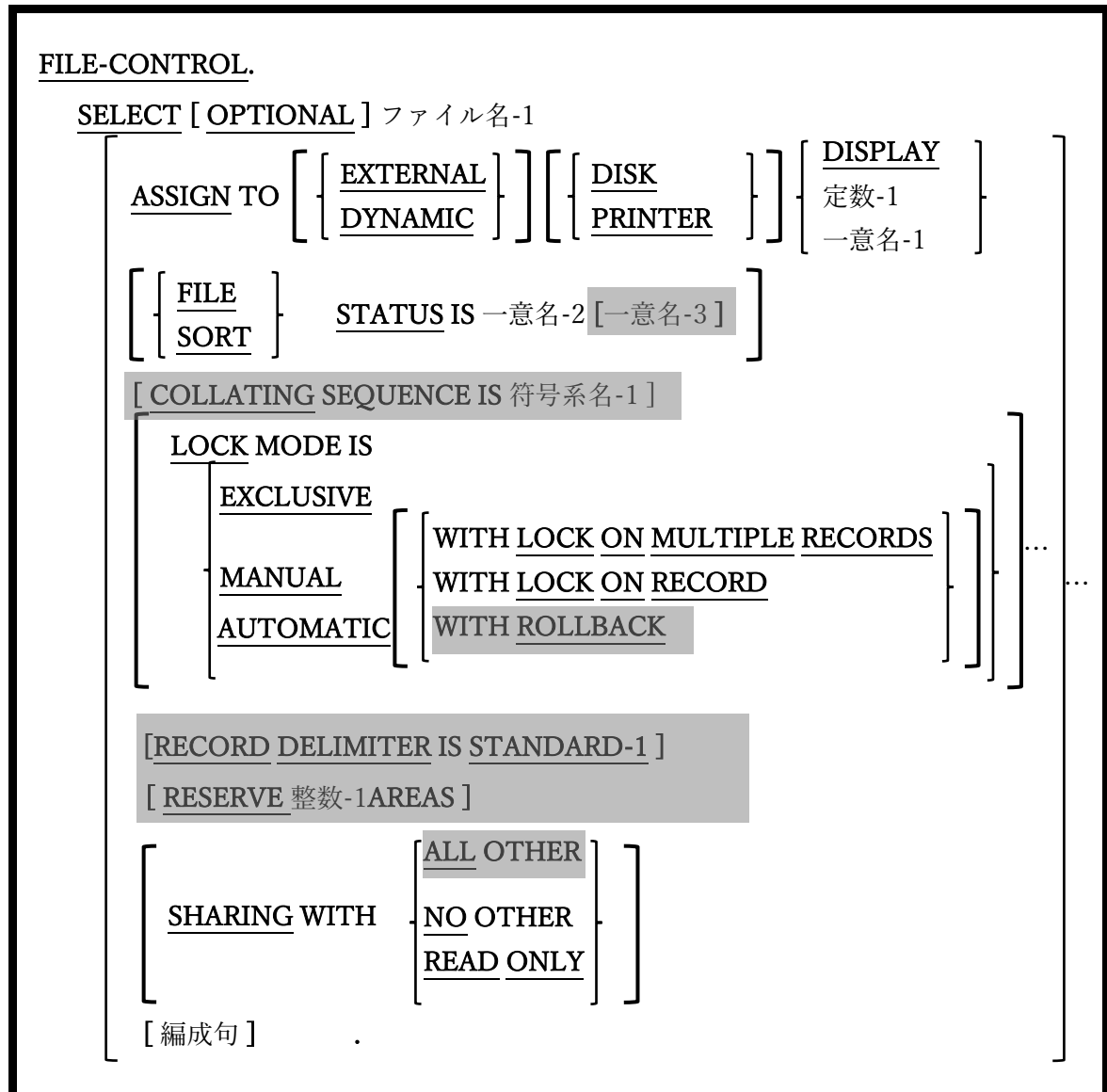
[I-O-CONTROL. 入出力管理記述]

入出力節(INPUT-OUTPUT SECTION)では、プログラムがアクセスするファイルを詳細に定義する。

1. 使用しているコンパイラの「config」ファイルの「relaxed-syntax-check」が「yes」に設定されている場合、入出力節のヘッダーを指定せずにファイル管理および入出力管理段落を指定することができる。構成ファイルやプログラムへの影響については 7.1.8 で説明する。

4.2.1. ファイル管理段落

図 4-10-ファイル管理段落構文



ファイル管理(FILE-CONTROL)段落の SELECT 文は、ファイル定義を作成し、外部オペレーティングシステム環境とリンクする。ここに示す例は、すべてのファイル形式に共通している SELECT 句である。次の節では、特定のファイル形式で用いる特別な SELECT 句について説明する。

1. COLLATING SEQUENCE、RECORD DELIMITER、RESERVE、SHARING WITH

ALL OTHER 句、および二次 FILE-STATUS 項目と LOCK MODE … WITH ROLLBACK の指定は、構文的には認識されるが、opensource COBOL では現在サポートされていない。

2. OPTIONAL 句は、プログラムに入力データを渡すために用いられるファイルにのみ使用され、ファイルの実行時に使用可能であるかどうかを示す。ファイルが存在しないときに OPTIONAL ファイルを開こうとすると(6.31)、ファイルが使用できないことを示す、致命的ではないが特別なファイルステータス値(表 4-11 のステータスコード 05 を参照)を受け取る。その後にファイルを読み取ろうとすると(6.33)、ファイル終了条件が返される。
3. opensource COBOL コンパイラパーサーテーブルは、実際にやや不合理な文がコーディングされても正常に解析できる。

```
SELECT My-File ASSIGN TO DISK DISPLAY.
```

効果としては、PC 画面に割り当てられたファイルを作成するためにコーディングされたものと同じ結果が得られる。

```
SELECT My-File ASSIGN TO DISPLAY.
```

4. ASSIGN 句で「定数-1」オプションを使用すると、COBOL ファイルからオペレーティングシステムファイルへの外部リンクが次のように定義される。
 - 「DD_定数-1」という名前の環境変数が存在する場合、その値はファイルのフルパスまたはファイル名として扱われる。そうでない場合は次へ。
 - 「dd_定数-1」という名前の環境変数が存在する場合、その値はファイルのフルパスまたはファイル名として扱われる。そうでない場合は次へ。
 - 「定数-1」という名前の環境変数が存在する場合、その値はファイルのフルパス

またはファイル名として扱われる。そうでない場合は次へ。

- 定数自体が、ファイルへのフルパスまたはファイル名として扱われる。

この動作は、プログラムのコンパイル時に用いる構成ファイルの「filename-mapping」設定の影響を受ける。上記の動作は、「filename-mapping : yes」が有効な場合にのみ適用され、「filename-mapping : no」に設定すると、最後のオプション(定数自体をフルファイル名として扱う)のみが可能となる。構成ファイルやプログラムへの影響については 7.1.8 で説明する。

一意名-2 の PICTURE(FILE STATUS 句)は 9(2)でなければならない。入出力ステータスコードは、ファイルに対して実行されるすべての入出力文の後に、この一意名に保存される。以下が、考えられるステータスコードの一覧である。

表 4-11-ステータスコード

ステータス値	意味
00	成功
02	成功(重複レコードキーが検出された)
05	成功(オプションファイルが存在しない)
07	成功(ユニットが存在しない)
10	ファイル終了
14	キー範囲外
21	キーが無効である
22	キーの値の重複が検出された
23	キーが存在しない
30	永続的入出力エラー
31	ファイル名に一貫性がない
34	ファイル区域外である
35	ファイルが存在しない
37	アクセス権拒否
38	ファイルがロックで閉じられている

opensource COBOL Programmers Guide	環境部
------------------------------------	-----

39	属性の矛盾が検出された
41	ファイルが既に開かれている
42	ファイルが開かれていない
43	読み込みが行われていない
44	レコードのオーバーフロー
46	読み込みエラー
47	OPEN INPUT が拒否された
48	OPEN OUTPUT が拒否された
49	OPEN I/O が拒否された
51	レコードがロックされている
52	ページ終了
57	LINAGE 指定が無効である
61	ファイル共有の失敗
91	ファイルが利用できない

5. LOCK 句と SHARING 句は、このファイルと同時に実行されている他のプログラムも、ファイルを使用できる条件を定義する。ファイルのロックと共有については、6.1.9 で説明する。

4.2.1.1. 順編成ファイル

図 4-12-順編成ファイルの指定

```
ORGANIZATION IS [ { RECORD BINARY  
                     LINE } ] SEQUENTIAL  
[ ACCESS MODE IS SEQUENTIAL ]  
[ PADDING CHARACTER IS { 定数-1  
                           一意名-1 } ]
```

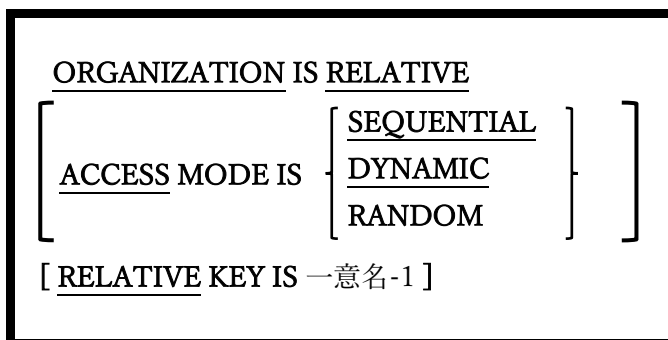
SEQUENTIAL ファイルとは、ファイル内のデータを順次処理することしかできない内部構造(COBOL では編成と呼ばれる)を持つファイルである。ファイルの 100 番目のレコードを読み取るには、レコードの 1 から始めて 99 までを読み取る必要がある。

1. ORGANIZATION RECORD BINARY SEQUENTIAL として宣言されたファイルは、明示的なレコード終了区切り文字順序のないレコードで構成される。ファイル内のレコードは、(レコード長に基づいて)計算されたバイトオフセットによって、ファイルに「書き出し」される。ファイルにはプログラムに区切り文字が埋め込まれているため、標準のテキスト編集ソフトウェアやワードプロセッシングソフトウェアでは作成できない。このようなファイルには、USAGE DISPLAY または USAGE COMPUTATIONAL(種類は任意である)のデータが含まれている可能性があり、これは文字順序がレコード終了の区切り文字として解釈されないためである。
2. ORGANIZATION IS RECORD BINARY SEQUENTIAL の指定と、ORGANIZATION SEQUENTIAL の指定は同じである。
3. ORGANIZATION LINE SEQUENTIAL として宣言されたファイルは、ASCII 改行文字(X "10")で終了するレコードで構成される。LINE SEQUENTIAL ファイルを読み取る場合、ファイルの FD で示されるサイズを超えた分のレコードは切り捨てられ、そのサイズより短いレコードは右側が PADDING CHARACTER 値によって埋められる。

4. PADDING CHARACTER が指定されていない場合は SPACE が指定されたものとみなす。
5. PADDING CHARACTER 句は、すべての ORGANIZATION ファイルで構文的には受け入れられるが、LINE SEQUENTIAL ファイルがレコードを埋めることができる唯一のファイルであるため意味を持つ。
6. 固定長と可変長、両方のレコード形式がサポートされている。
7. PRINTER または CONSOLE に ASSIGN されたファイルは、ORGANIZATION LINE SEQUENTIAL として指定する必要がある。
8. SEQUENTIAL ファイルの処理に関する文については、CLOSE(6.9)、COMMIT(6.10)、DELETE(6.13)、MERGE(6.27)、OPEN(6.31)、READ(6.33)、REWRITE(6.36)、SORT(6.40.1)、UNLOCK(6.48)およびWRITE(6.50)で説明する。

4.2.1.2. 相対編成ファイル

図 4-13-相対編成ファイルの指定



RELATIVE ファイルは、レコードを順次またはランダムに処理できる内部編成を持つファイルであり、ファイル内の相対レコード番号を指定することによって、レコードの読み取り、書き込み、および更新を行うことができる。

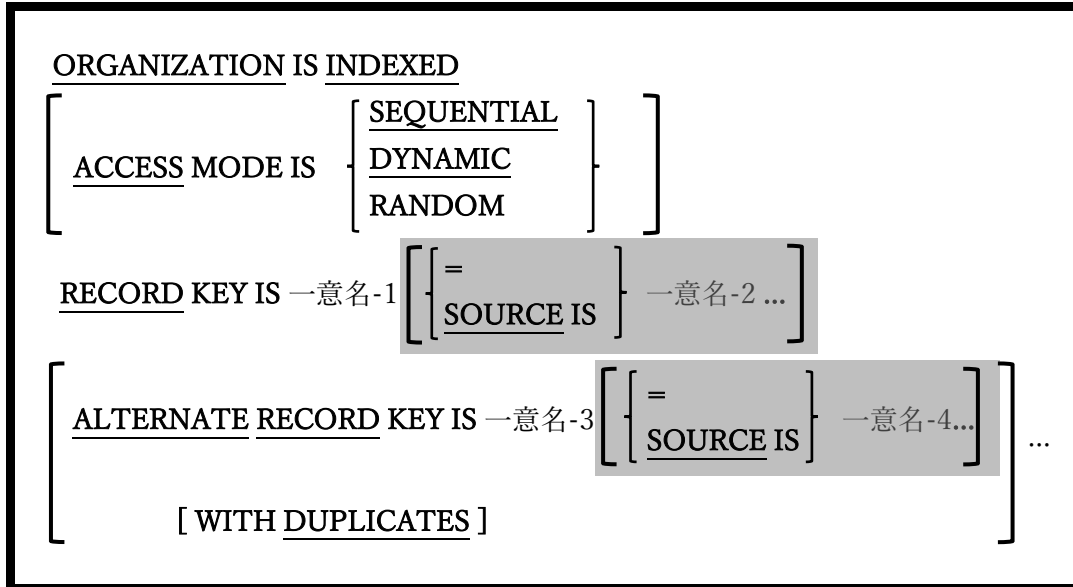
1. ORGANIZATION RELATIVE ファイルを CONSOLE または PRINTER に割り当てる

ことはできない。

2. RELATIVE KEY 句は、ACCESS MODE SEQUENTIAL が指定されている場合のみオプションとして扱う。
3. ORGANIZATION RELATIVE ファイルのレコードは可変長レコードを持つものとして定義できると考えられるが、ファイルは各レコードに対して最大レコード長を確保するように構造化される。
4. SEQUENTIAL の ACCESS MODE ではファイルのレコードが順次処理され、RANDOM の ACCESS MODE ではレコードがランダムに処理される。DYNAMIC ACCESS MODE では、ファイルが RANDOM または SEQUENTIAL モードのいずれかで処理され、プログラムの実行時に二つのどちらかを切り替えることができる(6.41 の START 文を参照)。
5. ACCESS MODE が指定されていない場合は SEQUENTIAL が指定されたものとみなす。
6. RELATIVE KEY データ項目は、ファイルのレコード内項目にできない数値データ項目である。SEQUENTIAL アクセスモードで処理されている RELATIVE ファイルの現在の相対レコード番号を返し、RANDOM アクセスモードで RELATIVE ファイルを処理するときに、読み取りまたは書き込みされる相対レコード番号を指定する検索キーとなる。
7. RELATIVE ファイルの処理に関する文については、CLOSE(6.9)、COMMIT(6.10)、DELETE(6.13)、MERGE(6.27)、OPEN(6.31)、READ(6.33)、REWRITE(6.36)、SORT(6.40.1)、START (6.41)、UNLOCK(6.48)および WRITE(6.50)で説明する。

4.2.1.3. 索引編成ファイル

図 4-14-索引編成ファイルの指定



RELATIVE ファイルのような INDEXED ファイルでは、レコードが順次またはランダムに処理される場合がある。ただし RELATIVE ファイルとは異なり、INDEXED ファイル内のレコードの実際の位置は、レコード内の一つ以上の英数字項目値に基づいている。

例えば、製品データを含む INDEXED ファイルは、製品識別コードをキーとして用いる場合がある。つまり、「A6G4328」番目のレコードまたは「Z8X7723」番目のレコードの製品 ID の値に基づいて、直接レコードを読み取り、書き込み、または更新することができる。

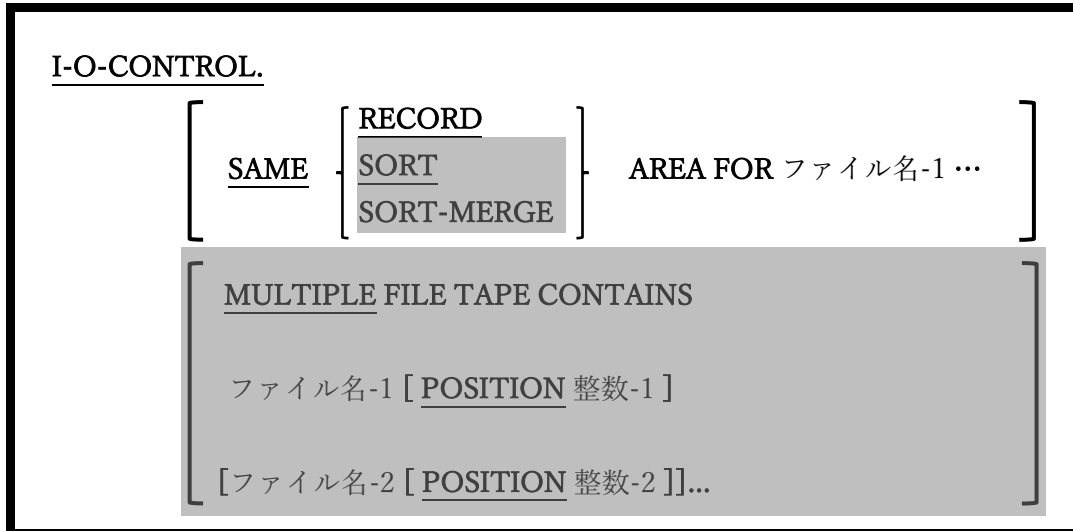
1. いわゆる「分割キー」の指定は構文的には認識されるが、現在 opensource COBOL ではサポートされていない。
2. SEQUENTIAL の ACCESS MODE では、ファイルのレコードが RECORD KEY または ALTERNATE RECORD KEY の値によって順次処理され、RANDOM の ACCESS MODE ではレコードがキー項目内でランダムに処理される。DYNAMIC ACCESS MODE では、ファイルが RANDOM または SEQUENTIAL モードのいずれかで処理され、プログラムの実行時に二つのどちらかを切り替えることができる (6.41 の

START 文を参照)。

3. ACCESS MODE が指定されていない場合は SEQUENTIAL が指定されたものとみなす。
4. RECORD KEY 句は、ファイル内レコードへ一次アクセスするために用いるレコード内の項目を定義する。
5. ALTERNATE RECORD KEY 句では、レコードに直接アクセスするための代替手段となるレコード内の追加項目、またはファイルの内容を順次処理できる追加項目を定義する。必要であれば、レコードに対して重複する代替キー値を許可することもできる。
6. 複数の ALTERNATE RECORD KEY 句があり、それぞれがファイルの代替キーを追加で定義している場合がある。
7. RECORD KEY 値はすべてのレコードにおいて一意でなければならない。ファイル内レコードの ALTERNATE RECORD KEY 値は、代替キーに WITH DUPLICATES 句が指定されている場合にのみ、重複する値を持つことが可能となる。
8. INDEXED ファイルの処理に関する文については、CLOSE(6.9)、COMMIT(6.10)、DELETE(6.13)、MERGE(6.27)、OPEN(6.31)、READ(6.33)、REWRITE(6.36)、SORT(6.40.1)、START (6.41)、UNLOCK(6.48)および WRITE(6.50)で説明する。

4.2.2. 入出力管理段落

図 4-15-入出力管理段落構文

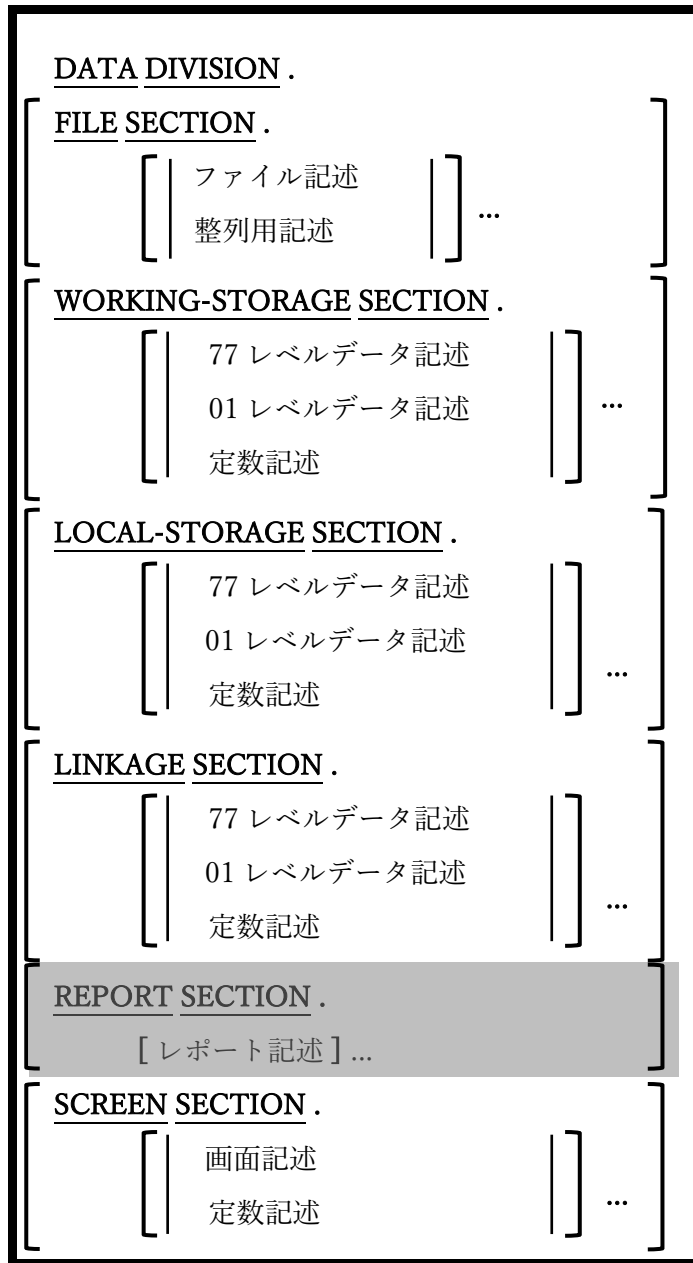


入出力管理(I-O-CONTROL)段落は、特定のファイル処理を最適化するために用いる。

1. SAME SORT AREA 句と SAME SORT-MERGE AREA 句は機能しないが、SAME RECORD AREA は機能する。
2. SAME RECORD AREA 句を使うと、複数のファイルが同一の入力および出力メモリバッファを共有するように指定できる。これらのバッファは巨大化してしまうことがあり、複数のファイルで同じバッファメモリを共有することによって、プログラムが使用するメモリ量の大幅な削減が可能となる(これにより手続き型コードまたはデータのための「空白」ができる)。この機能を使う場合は、指定したファイルが同時に開かないように注意することが必要である。
3. MULTIPLE FILE TAPE 句は廃止されたため、認識はされるがサポートはされていない。

5. データ部

図 5-1-データ部の形式



データ部(DATA DIVISION)は、プログラムが処理するすべてのデータを定義するために利用される。データ型やデータの使用方法に応じて、左に示した構文の骨組みからもわかるように、一つの節ごとに定義されている。

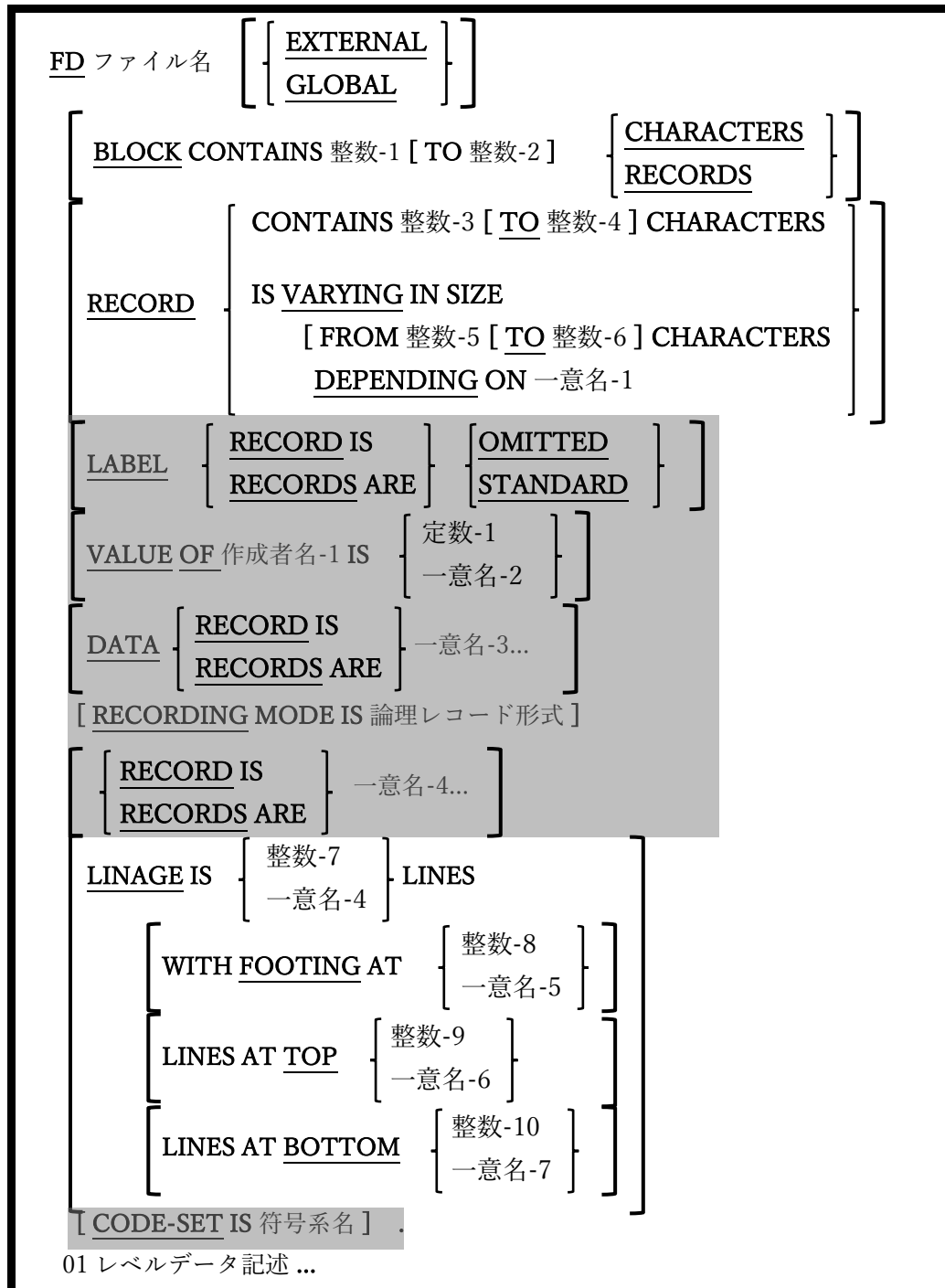
1. 宣言されているどの節も、提示されている順序で指定する必要がある。データ部が必要でない場合は、ヘッダー自体を省略することができる。
2. レポート節(REPORT SECTION)は構文的には認識されるが、利用すると対応されて

いないものとして拒否されてしまう。opensource COBOL は RWCS(レポート作成制御システム)に対応していないためである。(ただし、ファイル記述項では LINAGE 句がサポートされている)。

3. 局所場所節(LOCAL-STORAGE SECTION)は作業場所節(WORKING-STORAGE SECTION)と同じ方法で使用されるが、一つだけ例外がある。局所場所節で定義されたデータは、プログラム(ほとんどがサブプログラム)が実行される度に、初期状態に〔再〕初期化される。一方で、作業場所節のデータは静的であり、プログラムが中断されるか、メインプログラムの実行が終了するまで、最後に利用していた状態が保たれる。
4. 局所場所はネストされたプログラムでは使用できない。
5. 画面節(SCREEN SECTION)ではレポートの構造をレイアウトするレポート節を使う時と同様の規則や構文を使ったテキストベースでの画面レイアウトを定義できる。
6. opensource COBOL には共通場所節(COMMON-STORAGE SECTION)がないことに注意が必要である。実際に、この特徴は COBOL 規格から削除された。ただし機能的には、EXTERNAL または GLOBAL データ項目属性に置き換えられている。

5.1. ファイル記述

図 5-2-ファイル記述構文



プログラム内の SELECT で指定されたすべてのファイルについて詳細な記述が必要で、ファイル節(FILE SECTION)でコード化される。記述方法には、ファイル記述(FD)と整列用

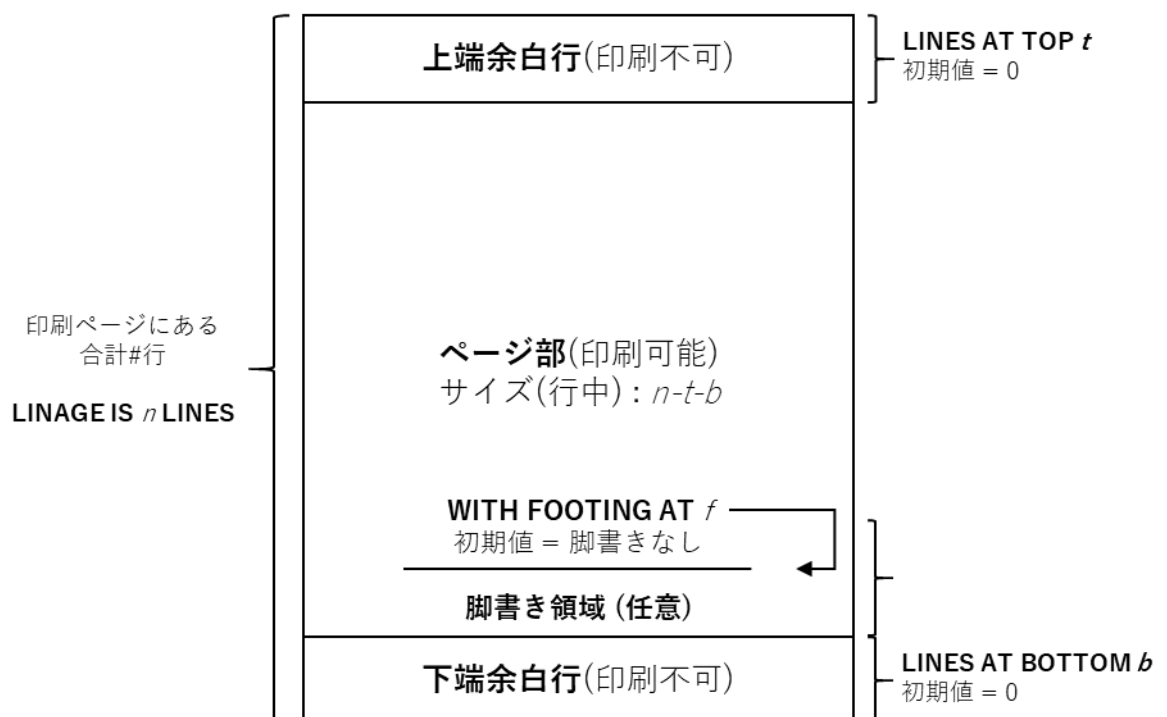
記述(SD)があり、それぞれ通常のデータファイルの記述と、作業ファイルの整列に使用される。ファイル記述では、ファイルで使用するレコード形式と、それらのレコードが効率的に処理を行うように、物理的ブロックに「まとめる」方法について詳細に説明する。

1. CODE-SET 句では、構文的に認識されているが、opensource COBOL では現時点でサポートされていない。
2. LABEL RECORD 句、DATA RECORD 句、RECORDING MODE 句、および VALUE OF 句は使われなくなった。使用しても生成されたコードに影響はない。DATA RECORD 句で指定された一意名はプログラム内で定義されているが、コンパイラの方は一意名が実際にファイルのレコードとして指定されているかどうかは問題にしない。
3. COBOL 言語は複数ある論理データレコードを、単体の物理データレコードに「ブロック」として入れることができる。メモリブロックが新しいレコードでいっぱいになった時、順次処理される出力ファイルに対して、実際に物理的書き込みが行われる(6.10 の COMMIT 文を参照)。同様にファイルを連続して読み取る場合、ファイルに対して生成された最初の READ 文は、最初の物理レコード(ブロック)を取得し、そこから最初の論理レコードが取得され、プログラムに送られる。次に生成された READ 文は、バッファが使い果たされるまで連続する論理コードを取得し、使い果たされると、次の物理レコードの取得のために別の物理的読み取りが実行される。ファイル記述の BLOCK CONTAINS 句を使用すると、プログラマに対して完全に透過的な方法ですべての処理を実行できる。
4. LINE SEQUENTIAL ファイルを使用する場合、RECORD CONTAINS 句と RECORD IS VARYING 句は無視される(警告メッセージが表示される)。他のファイル編成において、このような相互に排他的な句は、ファイル内のデータレコードの長さを定義していて、その長さはブロックのサイズを計算するために BLOCK CONTAINS ... RECORDS 句によって使用される。
5. REPORT IS 句は構文的に認識されているが、RWCS は opensource COBOL では現時

点でサポートされていないため、エラーが発生する。

6. LINAGE 句は、ORGANIZATION RECORD BINARY SEQUENTIAL または ORGANIZATION LINE SEQUENTIAL ファイルのみ指定できる。ORGANIZATION RECORD SEQUENTIAL ファイルで使用される場合、ファイル定義は暗黙的に LINE SEQUENTIAL に変更される。
7. LINAGE 句は図 5-3 からわかるように、印刷ページの様々な領域の論理的な境界線を (行数の観点から) 指定するために使用される。このページ構造の利用方法については、6.50(WRITE 文)で説明する。

図 5-3-LINAGE 句指定ページ構造

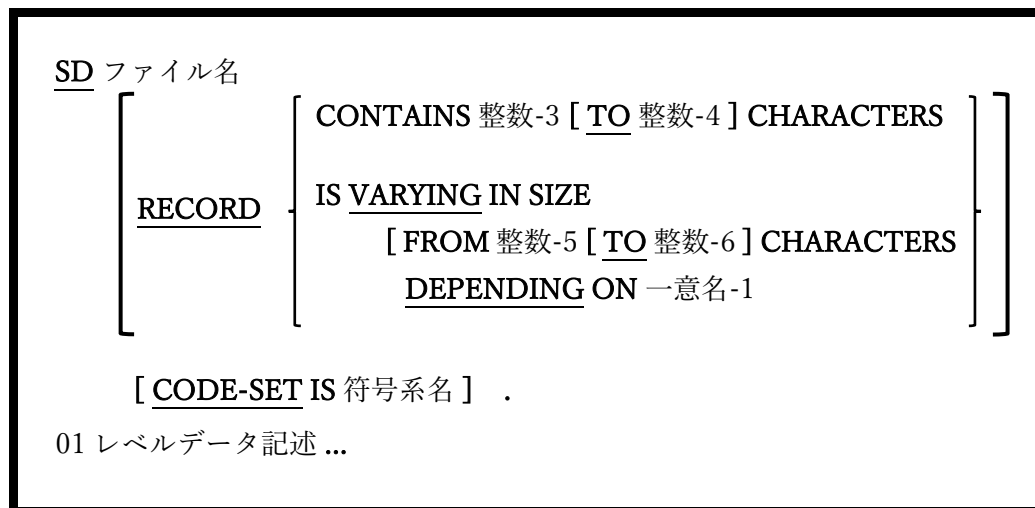


8. EXTERNAL 句を指定することにより、ファイル記述が必要な各コンパイルユニットで (EXTERNAL 句を使って) 記述されている場合、ファイル記述は、特定の実行スレッド内のすべてのプログラム (個別にコンパイルされるか、同じコンパイルユニットでコンパイルされる) 間で共有できる。この共有によって、異なる様々なプログラムでファイルを OPEN、読み書き、CLOSE することができる。

9. GLOBAL 句を指定することにより、ファイル記述が必要な各プログラムで(GLOBAL 句を使って)記述されている場合、ファイル記述は、特定の実行スレッド内の同じコンパイルユニットにあるすべてのプログラム間で共有できる。この共有によって、異なる様々なプログラムでファイルを OPEN、読み書き、CLOSE することができるが、個別にコンパイルされたプログラムは、GLOBAL ファイル記述を共有できない(ただし EXTERNAL ファイル記述は共有できる)。

5.2. 整列用記述

図 5-4-整列用記述段落



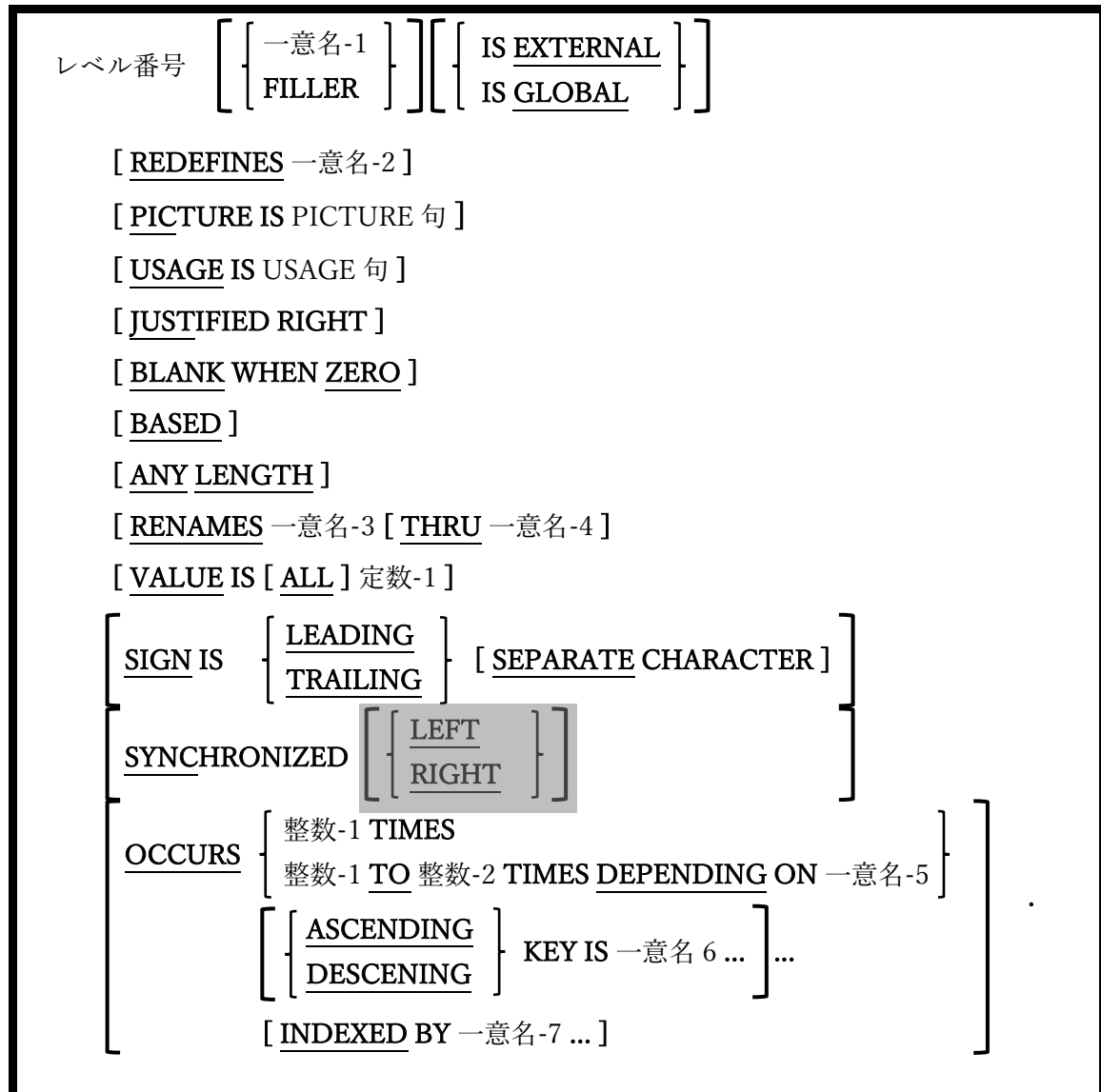
整列用ファイル(6.27 および 6.40.1 を参照)はファイル記述ではなく、整列用記述を使って説明する。

1. 完全な「ファイル記述(FD)」構文は実際には整列の記述に使用できるが、ここに示される構文要素のみ意味を持つことになる。
2. 整列用ファイルをディスクに割り当てる必要がある。
3. 整列されるデータの量が許容される場合、整列はメモリ内で実行される。

4. 一方でデータ量の確保にディスク作業ファイルが必要な場合、TMPDIR、TMP、または TEMP 環境変数で定義されたフォルダ内のディスクに自動で割り当てられる(7.2.4を参照)。これらのディスクファイルは、プログラムの実行が(通常またはその他の方法で)終了した場合、自動で削除されない。一時的な整列用作業ファイルは、自分自身から、または整列が終了した自分のプログラムから、ファイルを削除したい場合に「cob*.tmp」と命名される。
5. 整列用ファイルの SELECT 文で特定のファイル名を指定すると、そのファイル名は無視される。

5.3. データ記述の形式

図 5-5-データ記述の一般形式



ここで示した構文の骨組みは、画面節を除く、すべてのデータ部の節でデータ項目が定義される方法を提示している。

1. レベル番号の直後に一意名または FILLER を指定しない場合、FILLER を指定した場合と同じ動きをする。
2. 他の COBOL 実装と同様に、レベル番号は以下の値に制限されている。

- ・ 01－最上位レベルのデータ項目で、それ自体で完成している場合(基本項目とも呼ばれる)もあれば、従属項目に分割される場合(集団項目とも呼ばれる)もある。01レベルのデータ項目は「レコード」または「レコード記述」とよく呼ばれる。
- ・ 02-49－上位レベルのデータ項目の、従属部品であるデータ項目を定義するために使用されるレベル番号(レベル番号が数値的に小さいほど、定義されているデータ構造の階層全体で、データ項目は大きくなる－すべての構造化データは、単一の01レベルの項目から始める必要がある)。レベル 02-49 のすべてが基本項目でも良いし、レベル 02-48 がすべて集団項目でも良い。
- ・ 66－項目の再集団化－RENAMES 句は唯一このような項目を許可している。
- ・ 77－従属項目に分割されず、他のデータの従属項目でもないデータ項目(レベル 01 を使用しても同じことができるため、あまり使われない)。

この他にも特別な使い方をする二つのレベル番号(78 と 88)があるが、それは 5.5(78) と 5.4(88)でそれぞれ解説する。

3. レベル 66 のデータ項目は、すべてを参照できる集団項目名(一意名-1)を定義するように再集団化された構造内の、連続するデータ項目の再集団化にすぎない。
4. PICTURE 句は、定義されているデータ項目に含まれる可能性のあるデータのクラス(数値、アルファベット、または英数字)を定義する。また、データ項目用に予約されているストレージの容量も、(場合によっては USAGE 句と組み合わせて)定義する。基本的な 3 つのクラス定義 PICTURE 記号には以下の用途がある。

表 5-6-データのクラス定義 PICTURE 記号(9/A/X)

基本記号	意味・使用方法
9	1 桁の 10 進数用に予約されている場所を定義する。実際に占有されるストレージ量は、指定される USAGE 句によって異なる。
A	単一の英字(「A」-「Z」、 「a」-「z」)用に予約されている場所を定義する。各「A」は 1 バイトのストレージを表す。
X	1 文字のストレージ用に予約されている場所を定義する。各「X」は 1 バイトのストレージを表す。

以上の三つの記号は、PICTURE 句で繰り返し使用され、項目内に含まれる可能性のあるデータのクラス数を定義する。例：

- PIC 9999** 4 桁の正数を格納できるデータ項目を割り当てる(負の値については後述する)。項目の USAGE 句が DISPLAY 指定(既定値)の場合、4 バイトのストレージが割り当てられ、各バイトに「0」「1」「2」…「8」または「9」を入れることができる。数字限定というルールは実行時には強制されないが、コンパイル時にはルールに違反する定数値が項目に MOVE された場合、エラー警告が表示される。ランタイムエラーはクラスの条件テストを使用することで検出できる(6.1.4.2.2 を参照)。
- PIC 9(4)** 上記と同様一括弧で囲まれた繰り返し回数は、繰り返しを許可する任意の PICTURE 記号で利用できる。
- PIC X(10)** このデータ項目は任意の 10 文字(英数字形式)の文字列を格納できる。
- PIC A(10)** このデータ項目は任意の 10 文字(書式編集形式)の文字列を格納できる。文字のみが許可されるという強制はないが、エラーはクラスの条件テストを介して検出できる(6.1.4.2.2 を参照)。
- PIC AA9(3)A** X6 を指定するのと全く同じことだが、値を 2 文字、3 桁、1 文字の順にする必要があることを文書化している。文字の位置をチェックする「総

当たり攻撃」以外に、強制やエラー検出機能はない。

「A」 または「X」の PICTURE 記号を含むデータ項目は算術演算には使用できない。

上記に加え、表 5-7 は「PIC 9」データ項目で使える数値形式オプションの PICTURE 記号を示している。

表 5-7-数値形式オプションの PICTURE 記号(P/S/V)

数値形式の オプション記号	意味・使用方法
P	<p>実行時にデータ項目が参照されるとき 0 と見なされる、暗黙の桁位置を定義する。値の末尾に特定数の後続ゼロ(「P」につき 1 つ)が存在すると想定することによって、より少ないストレージを使用して、非常に大きい値を含んだデータ項目を割り当てられるように、この記号が使用される。</p> <p>このようなデータ項目に対して実行されるすべての演算およびその他の操作は、ゼロが実際に存在しているかのように動作する。</p> <p>値がそのような項目に格納されると、「P」記号で定義された桁位置は削除される。</p> <p>例えば、会社の今年の総収益に何百ドルもの収益を含んだデータ項目を割り当てる必要があるとする：</p> <p>01 Gross-Revenue PIC 9(9).</p> <p>このとき 9 バイトのストレージが予約され、値の 000000000～999999999 は総収益を表す。ただし、百万以下の単位が固定される場合(つまり後ろの 6 桁が常に 0 になる)、項目を次のように定義できる。</p> <p>01 Gross-revenue PIC 9(3)P(6).</p> <p>プログラム内で Gross-Revenue が参照されるときは必ず、ストレージ内の実際の値は、各 P 記号(この場合では全部で 6 つ)がゼロであるかのように扱われる。項目に 1 億 2800 万の値を格納するときは、「P」が「9」であるかのように扱う。</p>

opensource COBOL Programmers Guide	データ部
------------------------------------	------

	MOVE 128000000 TO Gross-Revenue.
S	PICTURE 値の最初の記号として使用する必要がある、このデータ項目では負の値が扱えることを示す。「S」がなければ、MOVE 文または算術文を介してデータ項目に格納された負の値からは、負の符号が取り除かれる(実際には絶対値となる)。
V	暗黙の小数点(存在する場合)が数値項目のどこにあるかを定義するために使用される記号。数値には小数点が 1 つしかないのと同じように、PICTURE 句には「V」が 1 つしかない。暗黙の小数点はストレージ内の空白を占有せずに、値の使用方法を指定する。例えば、値「1234」が PIC 999V9 として定義された項目のストレージ内にある場合、その値を参照するすべての文で「123.4」として扱われる。

5. USAGE DISPLAY の数値データにのみ許可される SIGN 句は、「S」記号の表現形式を指定する。SEPARATE CHARACTER 句の指定がないとき、データ項目の値の符号は、最終桁(TRAILING)または先頭桁(LEADING)を次のように変換することで符号化できる。

表 5-8-符号エンコード文字

最終/先頭 桁	正の数への 変換値	負の数への 変換値
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

SEPARATE CHARACTER 句が使用されている場合、実際の「+」または「-」記号が、先頭(LEADING)または最終(TRAILING)の文字として、項目の値に挿入される。

6. opensource COBOL は以下の表のように、「\$」、カンマ、アスタリスク(*)、小数点、CR、DB、+(プラス)、-(マイナス)、「B」、「0」(ゼロ)および「/」といった、すべての標準 COBOL PICTURE 編集記号を利用できる。

表 5-9-数字編集 PICTURE 記号

編集記号	意味・使用方法																					
-(マイナス)	<p>この記号は、PICTURE 句 の最初または最後に使用する必要がある。</p> <p>「-」を使用する場合、「+」、「CR」そして「DB」のいずれも使用することはできない。数字の編集に使用する。</p> <p>複数の「-」記号を連続して使用することは、項目の先頭でのみ許可される。これは浮動マイナス記号と呼ばれる。</p> <p>各「-」記号は、データ項目のサイズの 1 文字位置としてカウントされる。</p> <p>「-」記号が 1 つだけ指定されている場合、その記号は、項目に移動した値が負の場合は「-」に、そうでない場合は空白に「置き換え」られる。</p> <p>浮動マイナス記号が使用されている場合、編集プロセスは次のように機能すると考えること：</p> <ol style="list-style-type: none">1. 各「-」が実際には「9」である場合の編集値を決定する。2. 右端の「-」に対応する編集結果の数字を見つけ、その位置から編集値を左にスキャンしていき、左側に「0」文字しかない「0」に到達するまで続ける。3. 項目に移動した値が負の場合は「0」を「-」に、そうでない場合は空白に置き換える。4. その位置の左側にある残りの「0」文字をすべて空白で置き換える。 <p>例（記号␣は空白を表す）：</p> <table><tr><th>数値</th><th>PICTURE 句</th><th>結果</th></tr><tr><td>17</td><td>-999</td><td>␣017</td></tr><tr><td>-17</td><td>-999</td><td>-017</td></tr><tr><td>265</td><td>-----99</td><td>␣␣␣␣265</td></tr><tr><td>-265</td><td>-----99</td><td>␣␣␣-265</td></tr><tr><td>51</td><td>999-</td><td>051␣</td></tr><tr><td>-51</td><td>999-</td><td>051-</td></tr></table>	数値	PICTURE 句	結果	17	-999	␣017	-17	-999	-017	265	-----99	␣␣␣␣265	-265	-----99	␣␣␣-265	51	999-	051␣	-51	999-	051-
数値	PICTURE 句	結果																				
17	-999	␣017																				
-17	-999	-017																				
265	-----99	␣␣␣␣265																				
-265	-----99	␣␣␣-265																				
51	999-	051␣																				
-51	999-	051-																				

\$⁷

この記号は、「+」または「-」が PICTURE 句の左側に表示される場合を除き、その最初だけに使用する必要がある。数字の編集に使用する。

複数の「\$」記号を連続して使用することができ、*浮動通貨記号*と呼ばれる。

各「\$」記号は、データ項目のサイズの 1 文字位置としてカウントされる。

「\$」記号が 1 つだけ指定されている場合、項目値の有効桁数が多すぎて「\$」が占める位置が先頭のゼロ以外の数字を表す必要がある場合を除いて、その記号は編集値の位置に挿入される。この場合、「\$」は「9」として扱われる。

浮動通貨記号が使用されている場合、編集プロセスは次のように機能すると考えること：

1. 各「\$」が実際には「9」である場合の編集値を決定する。
2. 右端の「\$」に対応する編集結果の数字を見つけ、その位置から編集値を左にスキャンしていき、左側に「0」文字しかない「0」に到達するまで続ける。
3. 「0」を「\$」に置き換える。
4. その位置の左側にある残りの「0」文字をすべて空白で置き換える。

例(記号**␣**は空白を表す)：

数値	PICTURE 句	結果
17	\$999	\$017
265	\$\$\$\$99	␣␣␣\$265

⁷ デフォルトの通貨記号は「\$」であるが、他の国では異なる通貨記号を使用している。特殊名段落(4.1.4 を参照)では、任意の記号を通貨記号として定義することができる。例えば、通貨記号が「#」という文字に定義されている場合、「#」文字を PICTURE 編集記号として使用できる。

* (アスタリスク)

この記号は、「+」または「-」が PICTURE 句の左側に表示される場合を除き、その最初だけに使用する必要がある。数字の編集に使用する。

複数の「*」記号の連続した使用は、許可されているだけでなく、一般的な使用法である。これを浮動チェック保護記号と呼ぶ。

各「*」記号は、データ項目のサイズの 1 文字位置としてカウントされる。

編集プロセスは、次のように機能すると考えること：

1. 各「*」が実際には「9」である場合の編集値を決定する。
2. 右端の「*」に対応する編集結果の数字を見つけ、その位置から編集値を左にスキャンしていき、左側に「0」文字しかない「0」に到達するまで続ける。
3. 「0」を「*」に置き換える。
4. その位置の左側にある残りの「0」文字をすべて「*」に置き換える。

例：

数値	PICTURE 句	結果
265	*****99	****265

, (カンマ)⁸

PICTURE 文字列内の各カンマ(,)は、文字「,」が挿入される文字位置を表す。この文字位置は項目のサイズにカウントされる。「,」記号は、「,」文字の挿入を必要とする数字編集の桁数の精度が不十分である場合に、その左右にある浮動記号に見せかけることができる「スマート記号」である。

例(記号 b は空白を表す)：

数値	PICTURE 句	結果
17	\$\$,\$\$\$,\$99	bbbbbbb\$17
265	\$\$,\$\$\$,\$99	bbbbbbb\$265
1456	\$\$,\$\$\$,\$99	bbbb\$1,456

⁸ 特殊名段落で DECIMAL-POINT IS COMMA が指定されている場合、「.」と「,」の意味と使い方が反転する。

opensource COBOL Programmers Guide	データ部
------------------------------------	------

.(ピリオド) ⁸	<p>この記号は、暗黙の小数点が値に存在する位置で、編集値に小数点を挿入する。数字の編集に使用する。データ項目定義の最後に指定されたピリオドは、編集記号として扱われないことに注意すること！</p> <p>例：</p> <pre>01 Edited-Value PIC 9(3).99. 01 Payment PIC 9(3)V99 VALUE 152.19. ... MOVE Payment TO Edited-Value. DISPLAY Edited-Value.</pre> <p>152.19 が表示される。</p>
/(スラッシュ)	<p>この記号は、通常、印刷物の日付編集に使用され、編集値に「/」文字を挿入する。</p> <p>挿入された「/」文字は、編集結果で 1 バイトのストレージを占有する。</p> <p>例：</p> <pre>01 Edited-Date PIC 99/99/9999. ... MOVE 08182009 TO Edited-Date. DISPLAY Edited-Date.</pre> <p>08/18/2009 が表示される。</p>
+(プラス)	<p>この記号は、PICTURE 句の最初または最後に使用する必要がある。「+」を使用する場合、「-」、「CR」そして「DB」のいずれも使用することはできない。数字の編集に使用する。</p> <p>複数の「+」記号を連続して使用することは、項目の先頭でのみ許可される。これは浮動プラス記号と呼ばれる。</p> <p>各「+」記号は、データ項目のサイズの 1 文字位置としてカウントされる。</p> <p>「+」記号が 1 つだけ指定されている場合、その記号は、項目に移動した値が負の場合は「-」に、そうでない場合は「+」に「置き換え」られる。</p> <p>浮動マイナス記号が使用されている場合、編集プロセスは次のように機能すると考えること：</p> <ol style="list-style-type: none"> 1. 各「+」が実際には「9」である場合の編集値を決定する。 2. 右端の「+」に対応する編集結果の数字を見つけ、その位置から編集値を左にスキャンしていき、左側に「0」文字しかない「0」に到達するまで続ける。 3. 項目に移動した値が負の場合は「0」を「-」に、そうでない場合は

	<p>「+」に置き換える。</p> <p>4. その位置の左側にある残りの「0」文字をすべて空白で置き換える。</p> <p>例(記号 + は空白を表す)：</p> <table><tr><th>数値</th><th>PICTURE 句</th><th>結果</th></tr><tr><td>17</td><td>+999</td><td>+017</td></tr><tr><td>-17</td><td>+999</td><td>-017</td></tr><tr><td>265</td><td>+++++99</td><td>+265</td></tr><tr><td>-265</td><td>+++++99</td><td>+-265</td></tr><tr><td>51</td><td>999+</td><td>051+</td></tr><tr><td>-51</td><td>999-</td><td>051-</td></tr></table>	数値	PICTURE 句	結果	17	+999	+017	-17	+999	-017	265	+++++99	+ 265	-265	+++++99	+ -265	51	999+	051+	-51	999-	051-
数値	PICTURE 句	結果																				
17	+999	+017																				
-17	+999	-017																				
265	+++++99	+ 265																				
-265	+++++99	+ -265																				
51	999+	051+																				
-51	999-	051-																				
0(ゼロ)	<p>この記号は、編集値に「0」文字を挿入する。挿入された「0」文字は、編集結果で1バイトのストレージを占有する。</p> <p>例：</p> <pre>01 Edited-Phone-Number PIC 9(3)B9(3)B9(4) MOVE 5185551212 TO Edited-Phone-Number. DISPLAY Edited-Phone-Number. 518 555 1212 と表示される。</pre>																					
B	<p>この記号は、空白文字を編集値に挿入する。挿入された空白文字は、編集結果で1バイトのストレージを占有する。</p> <p>例：</p> <pre>01 Edited-Phone-Number PIC 9(3)B9(3)B9(4) MOVE 5185551212 TO Edited-Phone-Number. DISPLAY Edited-Phone-Number. 518 555 1212 と表示される。</pre>																					

CR	<p>この記号は、PICTURE 句の最後に使用する必要がある。「CR」を使用する場合、「-」、「+」そして「DB」のいずれも使用することはできない。数字の編集に使用する。</p> <p>1つの PICTURE 句で複数の「CR」記号を使用することはできない。</p> <p>「CR」記号は、データ項目のサイズで 2 文字の位置としてカウントされる。</p> <p>項目に移動した値が負の場合、文字「CR」が編集値に挿入される。それ以外の場合は、2つの空白が挿入される。</p> <p>例(記号 ␣ は空白を表す)：</p> <table><tr><th>数値</th><th>PICTURE 句</th><th>結果</th></tr><tr><td>17</td><td>99CR</td><td>17␣␣</td></tr><tr><td>-17</td><td>99CR</td><td>17CR</td></tr></table>	数値	PICTURE 句	結果	17	99CR	17 ␣␣	-17	99CR	17CR
数値	PICTURE 句	結果								
17	99CR	17 ␣␣								
-17	99CR	17CR								
DB	<p>この記号は、PICTURE 句の最後に使用する必要がある。「DB」を使用する場合、「-」、「+」そして「CR」のいずれも使用することはできない。数字の編集に使用する。</p> <p>1つの PICTURE 句で複数の「DB」記号を使用することはできない。</p> <p>「DB」記号は、データ項目のサイズで 2 文字の位置としてカウントされる。</p> <p>項目に移動した値が負の場合、文字「DB」が編集値に挿入される。それ以外の場合は、2つの空白が挿入される。</p> <p>例(記号 ␣ は空白を表す)：</p> <table><tr><th>数値</th><th>PICTURE 句</th><th>結果</th></tr><tr><td>17</td><td>99DB</td><td>17␣␣</td></tr><tr><td>-17</td><td>99DB</td><td>17DB</td></tr></table>	数値	PICTURE 句	結果	17	99DB	17 ␣␣	-17	99DB	17DB
数値	PICTURE 句	結果								
17	99DB	17 ␣␣								
-17	99DB	17DB								
Z	<p>この記号は、「+」または「-」が PICTURE 句の左側に表示される場合を除き、その最初だけに使用する必要がある。数字の編集に使用する。</p> <p>複数の「Z」記号の連続した使用は、許可されているだけでなく、一般的な使用法である。これを浮動ゼロサプレッションと呼ぶ。</p> <p>各「Z」記号は、データ項目のサイズの 1 文字位置としてカウントされる。</p> <p>編集プロセスは、次のように機能すると考えること：</p> <ol style="list-style-type: none">1. 各「Z」が実際には「9」である場合の編集値を決定する。2. 右端の「Z」に対応する編集結果の数字を見つけ、その位置から編集									

	値を左にスキャンしていき、左側に「0」文字しかない「0」に到達するまで続ける。 3. 「0」を空白に置き換える。 4. その位置の左側にある残りの「0」文字をすべて空白に置き換える。 例(記号 b は空白を表す) :		
	数値	PICTURE 句	結果
	17	Z999	b017
	265	ZZZZZ99	bbbb265

同じ PICTURE 句で、複数の編集記号を浮動方式で使用することはできない。

7. 編集記号を含む数値データ項目は、数値編集項目と呼ばれる。このようなデータ項目は、様々な算術文で値を受け取る場合があるが、同じ文でデータのソースとして使用することはできない。これに該当するのは、ADD 文(6.5)、COMPUTE 文(6.11)、DIVIDE 文(6.15)、MULTIPLY 文(6.29)、および SUBTRACT 文(6.44)である。
8. EXTERNAL 句を指定することにより、データ項目が各コンパイル単位で(EXTERNAL 句を使って)記述されている場合、定義されているデータ項目は、特定の実行スレッド内のすべてのプログラム単位(個別にコンパイルされるか、同じコンパイル単位でコンパイルされる)間で共有できる。
9. GLOBAL 句を指定することにより、データ項目は、各プログラム単位で GLOBAL 句を使って記述されている場合、そして GLOBAL 句を使用したすべてのプログラム単位が、GLOBAL 句を使用したデータ項目を定義する最初のプログラム単位内にネストされている場合、特定の実行スレッド内の同じコンパイル単位内のすべてのプログラム単位間で共有できる。プログラム単位のネストについては、2.1 で説明している。
10. EXTERNAL 句は、77 または 01 レベルでのみ指定できる。
11. EXTERNAL 項目にはデータ名(つまり一意名-1)が必要であり、その名前を FILLER にすることはできない。

12. EXTERNAL 句は、GLOBAL 句、REDEFINES 句、または BASED 句と組み合わせることはできない。
13. VALUE 句は、EXTERNAL データ項目、または EXTERNAL データ項目に従属するものとして定義されたデータ項目では無視される。
14. OCCURS 句は、複数回繰り返される表⁹と呼ばれるデータ構造を作成するため、次の例のように使用される。

05 QUARTLY-REVENUE OCCURS 4 TIMES PIC 9(7)V99.

以下のように割り当てられる。

QUARTLY-REVENUE(1)	QUARTLY-REVENUE(2)	QUARTLY-REVENUE(3)	QUARTLY-REVENUE(4)
--------------------	--------------------	--------------------	--------------------

各オカレンスは、上で示されている添字構文(括弧で囲まれた数字定数、算術式、または数値識別子)を使用して参照される。OCCURS 句は集団レベルでも使用でき、集団構造全体が次のように繰り返される。

05 X OCCURS 3 TIMES.
10 A PIC X(1) .
10 B PIC X(1) .
10 C PIC X(1) .

X(1)			X(2)			X(3)		
A(1)	B(1)	C(1)	A(2)	B(2)	C(2)	A(3)	B(3)	C(3)

表の詳細については、6.1.1(表の参照)、6.38(SEARCH)、6.40(SORT)、および以下の 28 項で説明する。

15. オプションの DEPENDING ON 句を OCCURS 句に追加することで、可変長テーブルを作成できる。このような表は、整数-2 で指定された最大サイズまで割り当てられる。

⁹ あなたもよく知っている他のプログラミング言語では、このような構造を *配列* と呼ぶ。

実行時、一意名-5 の値によって、アクセス可能な表の要素数が決まる。

16. レベル番号が 01、66、77、88 のデータ記述項には OCCURS 句を指定できない。
17. VALUE 句は、コンパイラによって生成されたプログラムオブジェクトコード内のデータ項目が占有するストレージに割り当てられる、コンパイル時の初期値を指定する。オプションの「ALL」句は英数字定数でのみ使用でき、データ項目が完全に埋まるまで必要に応じて値が繰り返される。以下は ALL を使用する場合と、使用しない場合の例である。
- ```
PIC X(5) VALUE "A" - 次の値を保持する "A", 空白, 空白, 空白, 空白
PIC X(5) VALUE ALL "A" - 次の値を保持する "A", "A", "A", "A", "A"
PIC 9(3) VALUE 1 - 次の値を保持する 001
PIC 9(3) VALUE ALL "1" - 次の値を保持する 111
```
18. ASCENDING KEY 句、DESCENDING KEY 句、および INDEXED BY 句については、6.39(SEARCH)で説明する。
19. BASED 句と ANY LENGTH 句を併用することはできない。
20. JUSTIFIED RIGHT 句は、アルファベット(PIC A)または英数字(PIC X)項目でのみ有効であり、データ項目の長さよりも短い値は、データ項目に MOVE されるときに右端に詰められ、空白で埋められる。
21. BASED 句で宣言されたデータ項目には、コンパイル時にストレージが割り当てられない。実行時に ALLOCATE 文を使用することによって領域を割り当て、(オプションで)項目を初期化する。
22. ANY LENGTH 属性で宣言されたデータ項目には、コンパイル時の固定長はない。この項目は、サブルーチン引数の説明としての機能であるため、連絡節でのみ定義することができる。ANY LENGTH 項目には、A、X、または 9 記号を 1 つだけ指定する

PICTURE 句が必要である。

23. BLANK WHEN ZERO 句を数値項目で使用すると、その項目に 0 の値が MOVE された場合、値が自動的に空白に変換される。

24. REDEFINES 句により、一意名-1 は一意名-2 と同じ物理ストレージ領域を占有するため、ストレージは(おそらく)異なる構造、そして異なる方法で定義される。REDEFINES 句を使用するには、次の条件がすべて満たされている必要がある。

- a. 一意名-2 のレベル番号は一意名-1 のレベル番号と同じでなければならない。
- b. 一意名-2(および一意名-1)のレベル番号は、66、77、78、または 88 にすることはできない。
- c. 「n」が一意名-2(および一意名-1)のレベル番号を表す場合、レベル番号「n」の他のデータ項目を、一意名-1 と一意名-2 の間に定義することはできない。
- d. 一意名-1 に割り当てられた合計サイズは、一意名-2 に割り当てられた合計サイズと同じでなければならない。
- e. 一意名-2 に OCCURS 句を定義することはできない。ただし、一意名-2 に従属する OCCURS 句で定義された項目が存在する場合がある。
- f. 一意名-2 に VALUE 句を定義することはできない。88 レベルの条件名を除き、一意名-2 に従属するデータ項目に VALUE 句を含めることはできない。

25. 次の表は、利用可能な USAGE 句をまとめたものである。

|                                    |      |
|------------------------------------|------|
| opensource COBOL Programmers Guide | データ部 |
|------------------------------------|------|

表 5-10-USAGE 句一覧

| USAGE 句                                             | 割り当て領域(バイト)                                                         | ストレージ形式          | 負の値                    | PIC | 類似 USAGE 句                     |
|-----------------------------------------------------|---------------------------------------------------------------------|------------------|------------------------|-----|--------------------------------|
| <u>BINARY</u>                                       | PICTURE 句の「9」の数と、プログラムのコンパイルに使用される構成ファイル(7.1.8)の「バイナリサイズ」設定によって異なる。 | 最互換性<br>—24 項参照  | PICTURE 句に「S」記号がある場合は可 | 可   | COMPUTATIONAL, COMPUTATIONAL-4 |
| <u>BINARY-CHAR</u> or <u>BINARY-CHAR SIGNED</u>     | 1 バイト                                                               | ネイティブ<br>—24 項参照 | 可                      | 不可  |                                |
| <u>BINARY-CHAR UNSIGNED</u>                         | 1 バイト                                                               | ネイティブ—24 項参照     | 不可—25 項参照              | 不可  |                                |
| <u>BINARY-C-LONG</u> or <u>BINARY-C-LONG SIGNED</u> | コンピュータの C 言語の「long」データ型と同じ量のストレージを割り当てる。通常は 32 ビットだが、64 ビットの場合もある。  | ネイティブ—24 項参照     | 可                      | 不可  |                                |
| <u>BINARY-C-LONG UNSIGNED</u>                       | コンピュータの C 言語の「long」データ型と同じ量のストレージを割り当てる。通常は 32 ビットだが、64 ビットの場合もある。  | ネイティブ—24 項参照     | 不可—25 項参照              | 不可  |                                |
| <u>BINARY-DOUBLE</u> or <u>BINARY-DOUBLE SIGNED</u> | 「従来の」ダブルワード(64 ビット)のストレージを割り当てる。                                    | ネイティブ—24 項参照     | 可                      | 不可  |                                |
| <u>BINARY-DOUBLE UNSIGNED</u>                       | 「従来の」ダブルワード(64 ビット)のストレージを割り当てる。                                    | ネイティブ—24 項参照     | 不可—25 項参照              | 不可  |                                |
| <u>BINARY-LONG</u> or <u>BINARY-LONG SIGNED</u>     | ワード(32 ビット)のストレージを割り当てる。                                            | ネイティブ—24 項参照     | 可                      | 不可  | SIGNED-LONG, SIGNED-INT        |
| <u>BINARY-LONG UNSIGNED</u>                         | ワード(32 ビット)のストレージを割り当てる。                                            | ネイティブ—24 項参照     | 不可—25 項参照              | 不可  | UNSIGNED-LONG, UNSIGNED-INT    |



|                                    |      |
|------------------------------------|------|
| opensource COBOL Programmers Guide | データ部 |
|------------------------------------|------|

|                                                      |                                                                                                   |                  |                        |    |                         |
|------------------------------------------------------|---------------------------------------------------------------------------------------------------|------------------|------------------------|----|-------------------------|
| <u>BINARY-SHORT</u> or<br><u>BINARY-SHORT SIGNED</u> | ハーフワード(16 ビット)のストレージを割り当てる。                                                                       | ネイティブ—24 項参照     | 可                      | 不可 | SIGNED-SHORT            |
| <u>BINARY-SHORT UNSIGNED</u>                         | ハーフワード(16 ビット)のストレージを割り当てる。                                                                       | ネイティブ—24 項参照     | 不可—25 項参照              | 不可 | UNSIGNED-SHORT          |
| <u>COMPUTATIONAL</u>                                 | PICTURE 句の「9」の数と、プログラムのコンパイルに使用される構成ファイル(7.1.8)の「バイナリサイズ」設定によって異なる。                               | 最互換性—24 項参照      | PICTURE 句に「S」記号がある場合は可 | 可  | BINARY, COMPUTATIONAL-4 |
| <u>COMPUTATIONAL-1</u>                               | ワード(32 ビット)のストレージを割り当てる。                                                                          | 単精度浮動小数点         | 可                      | 不可 |                         |
| <u>COMPUTATIONAL-2</u>                               | 「従来の」ダブルワード(64 ビット)のストレージを割り当てる。                                                                  | 倍精度浮動小数点         | 可                      | 不可 |                         |
| <u>COMPUTATIONAL-3</u>                               | PICTURE 句の「9」ごとに 4 ビットを割り当て、さらに符号用に(末尾の)4 バイト項目を割り当て、最も近いバイトに切り上げる。<br>SYNCHRONIZED RIGHT(27 項参照) | バック 10 進数—26 項参照 | PICTURE 句に「S」記号がある場合は可 | 不可 | PACKED-DECIMAL          |
| <u>COMPUTATIONAL-4</u>                               | PICTURE 句の「9」の数と、プログラムのコンパイルに使用される構成ファイル(7.1.8)の「バイナリサイズ」設定によって異なる。                               | 最互換性—24 項参照      | PICTURE 句に「S」記号がある場合は可 | 可  | BINARY, COMPUTATIONAL   |
| <u>COMPUTATIONAL-5</u>                               | PICTURE 句の「9」の数と、プログラムのコンパイルに使用される構成ファイル(7.1.8)の「バイナリサイズ」設定によって異なる。                               |                  | PICTURE 句に「S」記号がある場合は可 | 可  |                         |
| <u>COMPUTATIONAL-X</u>                               | プログラムのコンパイルに使用される構成フ                                                                              | 最互換性—24 項参       | PICTURE 句に「S」          | 可  |                         |

|                                    |      |
|------------------------------------|------|
| opensource COBOL Programmers Guide | データ部 |
|------------------------------------|------|

|                       |                                                                                                                                                                 |                  |                        |    |                 |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|------------------------|----|-----------------|
|                       | ファイル内の「1～8」の「バイナリサイズ」設定に従って、PICTURE 句の「9」の数に基づいてバイトを割り当てる。「バイナリサイズ」の値「1～8」がどのように機能するかについては、7.1.8 を参照すること。                                                       | 照                | 記号がある場合は可              |    |                 |
| <u>DISPLAY</u>        | PICTURE 句に基づく—PICTURE 句の X、A、9、ピリオド、\$、Z、0、*、S(SEPARATE CHARACTER が指定されている場合)、+、-、または B 記号ごとに 1 文字 <sup>10</sup> を割り当てる。DB または CR 記号が使用されている場合は、さらに 2 バイトを追加する。 | 文字               | PICTURE 句に「S」記号がある場合は可 | 可  |                 |
| <u>INDEX</u>          | ワード(32 ビット)のストレージを割り当てる。                                                                                                                                        | ネイティブ—24 項参照     | 不可                     | 不可 |                 |
| <u>NATIONAL</u>       | USAGE NATIONAL は、構文的には認識されるが、opensource COBOL ではサポートされていない。                                                                                                     |                  |                        |    |                 |
| <u>PACKED-DECIMAL</u> | PICTURE 句の「9」ごとに 4 ビットを割り当て、さらに符号用に(末尾の)4 バイト項目を割り当て、最も近いバイトに切り上げる。SYNCHRONIZED RIGHT(27 項参照)                                                                   | パック 10 進数—26 項参照 | PICTURE 句に「S」記号がある場合は可 | 不可 | COMPUTATIONAL-3 |

<sup>10</sup> この属性では、1 文字は 1 バイトと同じである。ただし、Unicode を使用する opensource COBOL システムを独自に構築した場合(可能性は低い)は 1 文字=2 バイトである。

|                                    |      |
|------------------------------------|------|
| opensource COBOL Programmers Guide | データ部 |
|------------------------------------|------|

|                        |                              |              |           |    |                                     |
|------------------------|------------------------------|--------------|-----------|----|-------------------------------------|
| <u>POINTER</u>         | ワード(32 ビット) のストレージを割り当てる。    | ネイティブ—24 項参照 | 不可        | 不可 |                                     |
| <u>PROGRAM-POINTER</u> | ワード(32 ビット) のストレージを割り当てる。    | ネイティブ—24 項参照 | 不可        | 不可 |                                     |
| <u>SIGNED-INT</u>      | ワード(32 ビット) のストレージを割り当てる。    | ネイティブ—24 項参照 | 可         | 不可 | BINARY-LONG-SIGNED, SIGNED-LONG     |
| <u>SIGNED-LONG</u>     | ワード(32 ビット) のストレージを割り当てる。    | ネイティブ—24 項参照 | 可         | 不可 | BINARY-LONG SIGNED, SIGNED-INT      |
| <u>SIGNED-SHORT</u>    | ハーフワード(16 ビット) のストレージを割り当てる。 | ネイティブ—24 項参照 | 可         | 不可 | BINARY SHORT SIGNED                 |
| <u>UNSIGNED-INT</u>    | ワード(32 ビット) のストレージを割り当てる。    | ネイティブ—24 項参照 | 不可—25 項参照 | 不可 | BINARY-LONG UNSIGNED, UNSIGNED-LONG |
| <u>UNSIGNED-LONG</u>   | ワード(32 ビット) のストレージを割り当てる。    | ネイティブ—24 項参照 | 不可—25 項参照 | 不可 | BINARY-LONG UNSIGNED, UNSIGNED-INT  |
| <u>UNSIGNED-SHORT</u>  | ハーフワード(16 ビット) のストレージを割り当てる。 | ネイティブ—24 項参照 | 不可—25 項参照 | 不可 | BINARY-SHORT UNSIGNED               |

26. バイナリデータは、「ビッグエンディアン」または「リトルエンディアン」形式で格納することができる。

ビッグエンディアンのデータ割り当てでは、バイナリ項目を構成するバイトについて、最下位バイトが右端のバイトとなるように割り当てられる。例えば、10 進数で 20 の値を持つ 4 バイトのバイナリ項目は、00000014(16 進表記で表示)として割り当てられるビッグエンディアンとなる。

リトルエンディアンのデータ割り当てでは、バイナリ項目を構成するバイトについて、最下位バイトが左端のバイトとなるように割り当てられる。例えば、10 進数で 20 の値を持つ 4 バイトのバイナリ項目は、14000000(16 進表記で表示)として割り当てられるリトルエンディアンとなる。

CPU はビッグエンディアン形式を「理解」できるため、コンピュータシステム間でバイナリストレージの「最互換性」形式となる。

一部の CPU—ほとんどの Windows PC で使用されている Intel/AMD i386/x64 アーキテクチャプロセッサなどは、リトルエンディアン形式で格納されたバイナリデータの処理を得意とする。この形式が上記システムでより効率的であるため、「ネイティブ」バイナリ形式と呼ばれる。

バイナリストレージの 1 つの形式(通常はビッグエンディアン)のみをサポートするシステムでは、「最効率的な形式」と「ネイティブ形式」は同義語である。

27. UNSIGNED 属性が明示的にコーディングされているバイナリデータ項目、または PICTURE 句に「S」記号がないバイナリデータ項目に、負の値を格納することはできない。このような項目に負の値を格納しようとする、実際には正の数であるかのように解釈される負の数のバイナリ表現が発生する。例えば、Intel または AMD プロセッサを実行しているコンピュータでは、バイナリ値として表される -3 の値は  $11111101_2$  になる。その値が USAGE BINARY-CHAR UNSIGNED 項目に格納されると、実際には  $011111101_2$  または 253 として解釈される。

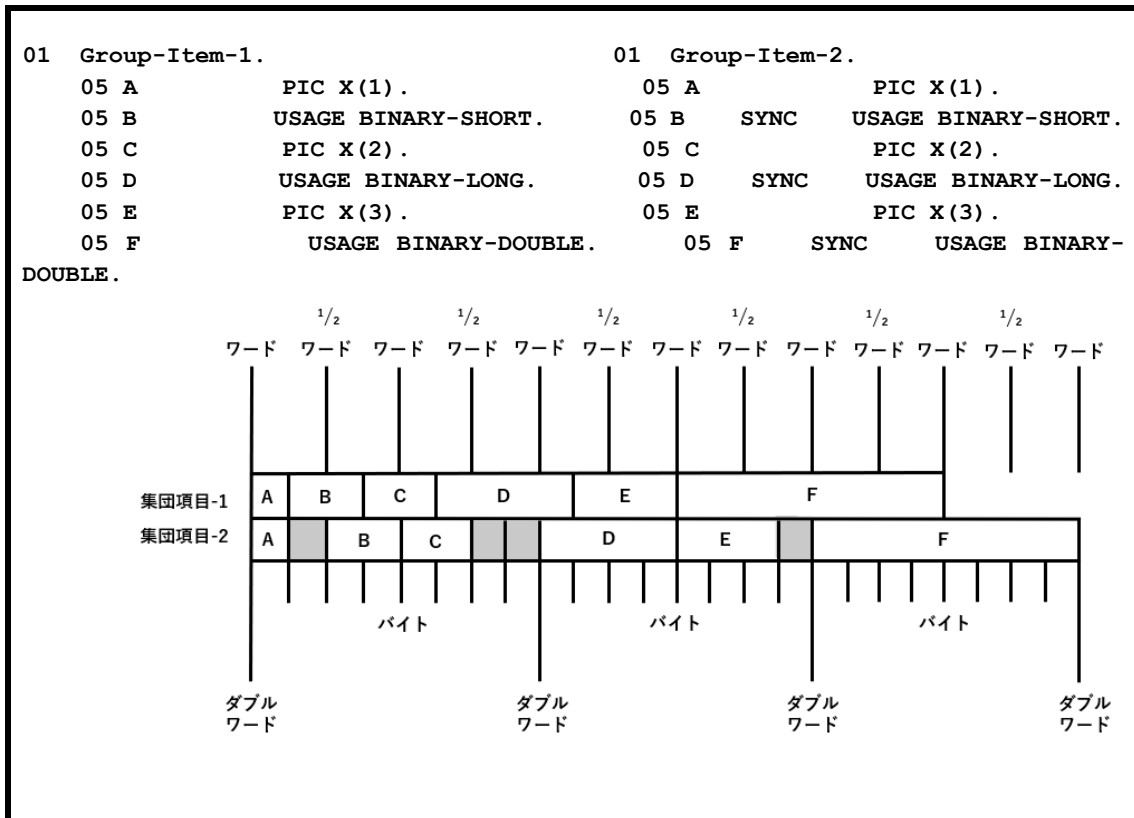
28. パック 10 進数(つまり、USAGE COMP-3 または USAGE PACKED-DECIMAL)データは、各バイトに 2 つの 4 ビット項目が含まれ、各項目が PICTURE 句の「9」を表し、10 進数 1 桁を格納する一連のバイトとして格納される。最後のバイトには、常に単一の 4 ビット数字(「9」に対応する)と 4 ビットの符号指示子(「S」記号が使用されていなくても常に存在する)が含まれる。最初のバイトには、PICTURE 句で使用された「9」記号の数に応じて、未使用の左端の 4 ビット項目が含まれる。符号指示子は、A から F までの 16 進数の値で、A、C、E、および F は正、B または D は負を示す。したがって、値が -15 の PIC S9(3) COMP-3 パック 10 進数項目は、16 進数の 015D(または 015B)が格納される。PICTURE 句に「S」が含まれていないパック 10 進数項目に負の数を格納しようとする、実際には負の数の絶対値が格納される。

29. SYNCHRONIZED 句(SYNC と省略される場合がある)は、バイナリ数値項目のストレージを最適化し、CPU のフェッチを可能な限り高速化して格納する。この同期は次のように実行される。

- a. バイナリ項目が 1 バイトのストレージを占有する場合、同期は実行されない。
- b. バイナリ項目が 2 バイトのストレージを占有する場合、バイナリ項目は次のハーフワード境界に割り当てられる。
- c. バイナリ項目が 4 バイトのストレージを占有する場合、バイナリ項目は次のワード境界に割り当てられる。
- d. バイナリ項目が 4 バイトのストレージを占有する場合、バイナリ項目は次のワード境界に割り当てられる。

次に示すのは、SYNCHRONIZED 句を使用する場合、そして使用しない場合の集団項目のストレージ割り当ての例である。

図 5-11 - SYNCHRONIZED 句の効果



灰色のブロックは、SYNC 句によって**集団項目-2** 構造に割り当てられた、未使用の「遊び」バイトを表す。

SYNCHRONIZED 句の LEFT および RIGHT オプションは、他の COBOL 実装との構文上の互換性のために認識はされるが、機能しない。

30. 表の初期化は、COBOL データ定義の難しい側面の 1 つである。基本的に 3 つの標準的な手法と、他の COBOL 実装に精通しているが opensource COBOL に慣れていない人にとっては興味深いと思われる 4 つ目の手法がある。以下の 3 つは「標準的な」手法である。

- a. コンパイル時に気にする必要はない。INITIALIZE 文を使用して、表の内のすべてのデータ項目オカレンスを(実行時に)、データ型固有の初期値(数値：0、英字および英数字：空白)に初期化する。

- b. 次のように、表の「親」として機能する集団項目に VALUE 句を含めることで、コンパイル時に小さな表を初期化する。

```

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. REDEFINES 句を使用して、コンパイル時にほぼすべてのサイズの表を初期化する。

```

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) .

```

c に示した表は、明らかに b よりも冗長である。しかし、c が優れている点は、より大きな表に必要な数の FILLER/VALUE 項目を記述できることである(そして、値は必要なだけ長くすることができる！)

多くの COBOL コンパイラでは、同じデータ項目で VALUE 句と OCCURS 句を使用することはできず、OCCURS 句に従属するデータ項目に VALUE 句を使用することもできない。一方で、opensource COBOL にはこれらの制限はない。次の例は、opensource COBOL で表を初期化する 4 番目の方法である。

```

05 X OCCURS 6 TIMES.
 10 A PIC X(1) VALUE "?".
 10 B PIC X(1) VALUE "%".
 10 N PIC 9(2) VALUE 10.

```

この例では、6 つの「A」項目が「?」、6 つの「B」項目が「%」、そして 6 つの「N」

項目が 10 に初期化される。この方法が役立つかわからないが、必要であれば使用できる。

## 5.4. 条件名

図 5-12-レベル 88 条件名記述構文

88 条件名-1

$$\left\{ \begin{array}{l} \underline{\text{VALUE IS}} \\ \underline{\text{VALUES ARE}} \end{array} \right\} \left\{ \text{定数-1} \left[ \left[ \begin{array}{l} \underline{\text{THROUGH}} \\ \underline{\text{THRU}} \end{array} \right] \text{定数-2} \right] \dots \right\} \dots$$

[ WHEN SET TO FALSE IS 定数-3 ]

条件名はブーリアン型(つまり「TRUE」/「FALSE」)のデータ項目である。

1. 条件名は常に別のデータ項目に従属して定義される。データ項目は基本項目である必要はない。
2. また、ストレージを占有しない。
3. 条件名に指定された VALUE(s)は、条件名の値を TRUE にする親要素データ項目の特定の値、および/または、値の範囲を指定する。
4. オプションの FALSE 句は、SET 文を使用して条件名-1 を FALSE に設定した場合に、親の基本データ項目に割り当てられる明示的な値を定義する。SET 文を使用して、条件名の TRUE/FALSE 値を指定する方法については、6.39.6 で詳しく説明する。
5. 条件名については、6.1.4.2.1 でも説明する。



## 5.5. 定数記述

図 5-13-78 レベル定数記述構文

78 一意名-1 VALUE IS 定数-1.

01 一意名-2 CONSTANT [ IS GLOBAL ] AS { 定数-2  
LENGTH OF 一意名-3  
BYTE-LENGTH OF 一意名-4 } .

この形式のデータ項目は、実際にストレージを割り当てることはないが、その代わりに、名前を英数字または数字定数に関連付ける役割がある。

1. 定数値を定義する場合において、二つの形式は基本的に同じであるが、「01 CONSTANT」を使用した場合にのみ、値が別の項目の長さである定数を定義することが可能である。
2. GLOBAL 句は構文的には認識されるが、現時点では opensource COBOL でサポートされていないため、コンパイラ警告が表示される。しかし、2009 年 2 月 6 日の opensource COBOL1.1 パッケージ化の時点では、実際にコンパイラを中断させる可能性がある。

## 5.6. 画面記述

図 5-14-画面節データ項目記述構文

レベル番号

[ { 一意名-1  
 FILLER } ]

[ JUSTIFIED RIGHT ]

[ BLANK WHEN ZERO ]

[ OCCURS 整数-1 TIMES ]

```

[BELL | BEEP]
[AUTO | AUTO-SKIP | AUTOTERMINATE]
[UNDERLINE]
[OVERLINE]
[SECURE]
[REQUIRED]
[FULL]
[PROMPT]
[REVERSE-VIDEO]
[BLANK LINE | SCREEN]
[ERASE EOL | EOS]

[SIGN IS { LEADING } [SEPARATE CHARACTER]
 { TRAILING }
[LINE NUMBER IS [PLUS] { 整数-2 }
 { 一意名-2 }]
[COLUMN NUMBER IS [PLUS] { 整数-3 }
 { 一意名-3 }]
[BACKGROUND-COLOR IS { 整数-4 } [{ HIGHLIGHT }]
 { 一意名-4 } [{ LOWLIGHT }]]
[BACKGROUND-COLOR IS { 整数-5 } [BLINK]
 { 一意名-5 }]
[{ PICTURE IS PICTURE 句 { USING 一意名-6 }
 { FROM { 一意名-7 } }
 { TO 一意名-8 }]
[VALUE IS 定数-1]

```

上に示した構文の枠組みは、画面節でデータ項目がどのように定義されているかを表す。これらのデータ項目は、特別な形式の ACCEPT 文(6.4)および DISPLAY 文(6.14.4)を介して使用され、TUI(「テキストユーザインターフェース」プログラム)を作成する。

1. レベル番号 66、78 および 88 のデータ項目は画面節で使用でき、他のデータ部節と同じ構文、規則、使用方法である。
2. BELL 句または BEEP 句(どちらも同義語である)を利用して、画面項目が表示されているとき可聴音を鳴らす。
3. AUTO 句(三つある形式はすべて同じ)は、AUTO 句のある項目が完全に入力されているとき、次の入力可能項目へと自動で進むカーソルが表示される。
4. UNDERLINE 句と OVERLINE 句は、現時点では Windows のコンソールウィンドウ API でサポートされていないため、Windows システムでは基本的に機能しない。しかし UNDERLINE 句は、FOREGROUND-COLOR 属性によって指定された(または暗黙の)値に関係なく、項目の前景色を青に表示する効果がある。これらの句が UNIX システムで機能するか否かは、使用する出力端末のビデオ属性によって異なる。
5. SECURE 属性は、データ入力(USING または TO)を許可する項目でのみ使用できる。この属性によって、項目に入力されたデータはすべて、アスタリスクとして表示される。
6. REQUIRED 属性と FULL 属性は、構文的には適切であるが、機能はしない。
7. PROMPT 属性は、すべての入力項目の既定の動作となっているため、opensource COBOL では不要である。<sup>11</sup>
8. REVERSE-VIDEO 属性は、指定または暗黙の FOREGROUND-COLOR 属性と BACKGROUND-COLOR 属性の意味を逆にする。
9. BLANK 句は、データ項目の LINE 句や COLUMN 句で示されたポイントから、画面

---

<sup>11</sup> PROMPT 属性は、非空白文字でマークすることで表示されるようにした、空の入力項目の指定に使用される。この機能は、opensource COBOL における編集可能なすべての画面項目で常に有効になっている(空白に下線を引いた文字が使用されている)。

または行を空白にする。さらに、コンソールウィンドウの前景色と背景色は、項目で指定されている色に設定される。レベル 01 項目(または従属項目)内でこの句を使用すると、その項目内に表示されるすべての項目が非表示になる。

10. ERASE 句は、コンソールウィンドウの最新行(EOL)または画面(EOS)の残りの部分を消去する。ERASE 句が消去したり、前景色と背景色を設定する項目の最後の方から始めていき、ERASE 句を含む項目に対して有効である。
11. LINE 句または COLUMN 句がない場合、画面節項目は画面項目を表す ACCEPT 文または DISPLAY 文によって、指定もしくは暗示される縦/横座標で始まるコンソールウィンドウに表示される。項目がコンソールウィンドウに表示された後、次の項目がその直後に表示される。

LINE 句と COLUMN 句は、コンソールウィンドウのどこに項目を表示するかを明示的に示す手段を提供する。座標は、絶対座標(「縦 1 横 5」)または以前に提示された項目の終わりに基づく相対座標(「縦+2 横+1」)で表すことができる。一意名や定数を使用して、絶対位置または相対位置を定義できる。一意名を使用する場合は、記号を編集しない PIC 9 項目である必要がある(COMPUTATIONAL-1 または COMPUTATIONAL-2 を除く、任意の数値 USAGE が許可される。浮動小数点 USAGE 仕様はそのどちらかは受け入れられるが、予測できない結果になることに注意)。

もちろん、LINE 句と COLUMN 句を使用せずに画面項目の暗黙的配置に依存している場合を除いて、項目は表示された縦/横の順序で定義する必要はない。

TAB キーと BACK-TAB(Shift-TAB)キーは、画面節で定義された順序に関係なく、コンソールウィンドウ上に項目が出現する縦/横の順序で、項目から項目へカーソルを配置する。

必要に応じて COLUMN は COL に省略が可能である。

12. FOREGROUND-COLOR 句と BACKGROUND-COLOR 句は、テキスト(前景)または画面(背景)の色を指定するために使用される。以下のような番号(0~7)によって色を指定する。

表 5-15-番号によって指定される画面色

| 整数 | 色  |
|----|----|
| 0  | 黒  |
| 1  | 青  |
| 2  | 緑  |
| 3  | 青緑 |
| 4  | 赤  |
| 5  | 赤紫 |
| 6  | 黄  |
| 7  | 白  |

13. HIGHLIGHT および LOWLIGHT オプションは、テキストの輝度(前景)を制御する。これは 3 レベルの強度方式(LOWLIGHT、指定なし、HIGHLIGHT)の提供を目的としているが、Windows のコンソールは 2 レベルまでをサポートしているため、LOWLIGHT はこの句を完全に省略した場合と同じである。この修飾子を FOREGROUND-COLOR 属性に使用すると、次の表のように実際には 8 色だけでなく 16 色のテキストを使用できる。

表 5-16- LOWLIGHT/ HIGHLIGHT オプションによる画面色

| FOREGROUND-COLOR 整数 | LOWLIGHT | HIGHLIGHT |
|---------------------|----------|-----------|
| 0                   | 黒        | 暗灰        |
| 1                   | 暗青/藍     | 明青        |
| 2                   | 暗緑       | 明緑        |
| 3                   | 暗青緑      | 明青緑       |
| 4                   | 暗赤       | 明赤        |
| 5                   | 暗赤紫      | 明赤紫       |
| 6                   | 金/茶      | 黄         |
| 7                   | 明灰       | 白         |

14. BLINK 属性は、BACKGROUND-COLOR 仕様の外観を変更する。Windows のコンソールは点滅をサポートしていないため、Windows 版 opensource COBOL における BLINK の視覚効果は、LOWLIGHT/ HIGHLIGHT と組み合わせた FOREGROUND-COLOR において可能であるのと同様の 16 色を BACKGROUND-COLOR パレットに提供することである。

15. 前景色と背景色の属性は、他の項目から継承できる。前の項目からではなく、親のデータ項目(数値的に低いレベルのデータ項目)から継承される。以下の点に注意が必要である。

```

78 Black VALUE 0.
78 Blue VALUE 1.
78 Green VALUE 2.
78 White VALUE 7.
...
02 XYZ BACKGROUND-COLOR Black FOREGROUND-COLOR Green ...
 05 ABC BACKGROUND-COLOR Blue FOREGROUND-COLOR White ...
 05 DEF (no BACKGROUND-COLOR or FOREGROUND-COLOR specified) ...
DEF 項目の色は緑と白になる(XYZ から継承される)

```

16. VALUE 句は変更できない固定のテキストを定義するために使用される。

17. FROM 句は指定された定数または一意名から、内容を取得する必要がある項目を定義

するために使用される。

18. TO 句は初期値のないデータ入力項目を定義するために使用される。値を入力すると、指定した一意名に保存される。

19. USING 句は「FROM 一意名」と「TO 一意名」の組み合わせである。

## 6. 手続き部

### 6.1. 構成要素

#### 6.1.1. 表の参照

COBOL は括弧を使用して、表記述項を参照するための添字を指定する (COBOL の表は、他のプログラミング言語で配列と呼ばれる)。

4 列×3 行の文字グリッドを表す、以下のデータ構造を例に見てみよう：

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

次の図で網掛けされている GRID-CHARACTER は、

|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

次のコードで参照できる。

```
GRID-CHARACTER (2, 3)
```

添字は、数値(整数)定数、PIC 9(整数)データ項目、USAGE INDEX データ項目、またはこれらの任意の組み合わせを含む整数値をもたらす算術式として指定できる。算術式を表(配列)の添字として使用する機能は、多くの言語の場合で一般的となっているが、COBOL では稀である。



算術式については 6.1.4.1 で説明する。

### 6.1.2. データ名の修飾

COBOL では、データ名をプログラム内で複製することができ、修飾と呼ばれるプロセスを通じてデータ名の参照を一意にするという方法によって、データ名への参照を行うことができる。

動作中の修飾を確認するには、COBOL プログラムで定義された 2 つのデータレコードの、次のようなセグメントを確認する：

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

それでは、従業員の輸送先住所の CITY の部分を「Philadelphia」に設定してみる。明らかにコンパイラは、参照している 2 つの CITY 項目のどちらかを判別できなくなるため、以下の例は機能しない：

```
MOVE "Philadelphia" TO CITY.
```

この問題を解決するために、CITY の参照を次のように修飾できる。

```
MOVE "Philadelphia" TO CITY OF MAILING-ADDRESS.
```

残念ながら、どの CITY が参照されているかを具体的に判別するにはまだ不十分である。  
特定の CITY を正確に判別するには、次のようにコーディングする必要がある。

```
MOVE "Philadelphia" TO CITY OF MAILING-ADDRESS OF EMPLOYEE.
```

これによって、どの CITY が変更されているかについての混乱が生じることはなくなる。  
しかしもっと簡単な記述にすることもできる。COBOL では中間の修飾を省略できるため、以下のようなコーディングが可能である。

```
MOVE "Philadelphia" TO CITY OF EMPLOYEE.
```

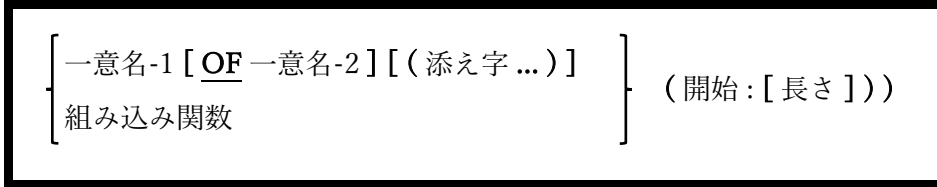
テーブルへの参照を修飾する場合は次のように記述する。

```
一意名-1 OF 一意名-2 (添え字…)
```

予約語の「IN」は「OF」の代わりとして使うことができる。

### 6.1.3. 部分参照

図 6-1-部分参照構文



COBOL'85 標準では、データ項目の一部のみへの参照を容易にするための部分参照の概念が導入された。opensource COBOL は、参照の修飾を完全にサポートしている。

開始値は、参照される開始文字位置を示し(文字位置の値は、一部のプログラミング言語は 0 から始まるが、この場合は 1 から始める)、長さは必要な文字数を指定する。長さが指定されていない場合、最初から最後までの子の文字位置に相当する値が想定される。

ここでいくつか例を挙げる。

|                                    |                                                                                                               |
|------------------------------------|---------------------------------------------------------------------------------------------------------------|
| <b>CUSTOMER-LAST-NAME (1:3)</b>    | CUSTOMER-LAST-NAME の最初の 3 文字を参照する。                                                                            |
| <b>CUSTOMER-LAST-NAME (4:)</b>     | CUSTOMER-LAST-NAME の 4 番目以降のすべての文字位置を参照する。                                                                    |
| <b>FUNCTION CURRENT-DATE (5:2)</b> | 現在の月を参照する(詳細については 6-13 ページの「CURRENT-DATE 組み込み関数」で説明する)。                                                       |
| <b>Hex-Digits (Nibble + 1:1)</b>   | 「Nibble」が 0~15 の範囲の値を持つ数値データ項目で、かつ Hex-Digits が「0123456789ABCDEF」の値を持つ PIC X(16)項目であるとする、与えられた数値を 16 進数に変換する。 |

**Array-Element (6) (7:5)**

Array-Element の 6 番目の配列の 5 文字を参照する。このとき文字位置は 7 から開始する。

参照の修飾は、MOVE 文、STRING 文、ACCEPT 文などの受け取り項目としても機能するなど、一意名が有効な場所であればどこでも使用できる。

**6.1.4. 式**

opensource COBOL は他の COBOL 実装と同様に、基本となる 2 つの式をサポートする。

- 数値結果を計算する「算術式」
- TRUE または FALSE 値を計算する「条件式」

0 や -1 などの算術値が、それぞれ FALSE や TRUE を表す他のプログラミング言語とは違い、COBOL は論理的な TRUE/FALSE 値と 0/-1 を異なるものとして扱う。opensource COBOL はこのポリシーに準拠している。

#### 6.1.4.1. 算術式

算術式は、次の演算子を使用して形成される。複数の演算子で構成される複雑な式では、演算の優先順位が適用され、優先順位の低い演算より高い演算の方が先行して計算される。

| 優先順位<br>演算子                                                                                                                                                                                                                                                       | 意味                                                                                |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| <p>図 6-2-符号(-)<br/>1 番目(最上位)</p> <div> <math display="block">- \left[ \begin{array}{l} \text{数値定数-1} \\ \text{一意名-1} \\ \text{(算術式-1)} \end{array} \right]</math> </div>                                                                                          | <p>単項減算演算子(-)は引数の算術否定を返す。引数と数値定数の-1 を掛けた値を有効値とする。</p>                             |
| <p>図 6-3-符号(+)<br/>1 番目(最上位)</p> <div> <math display="block">+ \left[ \begin{array}{l} \text{数値定数-1} \\ \text{一意名-1} \\ \text{(算術式-1)} \end{array} \right]</math> </div>                                                                                          | <p>単項加算演算子(+)は引数の値を返す。引数と数値定数の+1 を掛けた値を有効値とする。</p>                                |
| <p>図 6-4-べき乗演算子<br/>2 番目</p> <div> <math display="block">\left[ \begin{array}{l} \text{数値定数-1} \\ \text{一意名-1} \\ \text{(算術式-1)} \end{array} \right] ** \left[ \begin{array}{l} \text{数値定数-2} \\ \text{一意名-2} \\ \text{(算術式-2)} \end{array} \right]</math> </div> | <p>演算子の左側の引数の値を、右側の引数で示されるべき乗で計算する。opensource COBOL では「**」記号の代わりに「^」記号が使用できる。</p> |

|                                                                                                                                                                                                                                                                                                                                             |                        |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------|
| <p>図 6-5-乗算演算子<br/>3 番目</p> <div style="border: 2px solid black; padding: 10px; display: inline-block;"> <math display="block">\left[ \begin{array}{c} \text{数値定数-1} \\ \text{一意名-1} \\ \text{(算術式-1)} \end{array} \right] * \left[ \begin{array}{c} \text{数値定数-2} \\ \text{一意名-2} \\ \text{(算術式-2)} \end{array} \right]</math> </div>      | 演算子の左右の引数の乗算を求める。      |
| <p>図 6-6-除算演算子<br/>3 番目</p> <div style="border: 2px solid black; padding: 10px; display: inline-block;"> <math display="block">\left[ \begin{array}{c} \text{数値定数-1} \\ \text{一意名-1} \\ \text{(算術式-1)} \end{array} \right] / \left[ \begin{array}{c} \text{数値定数-2} \\ \text{一意名-2} \\ \text{(算術式-2)} \end{array} \right]</math> </div>      | 演算子の左右の引数の除算を求める。      |
| <p>図 6-7-加算演算子<br/>4 番目(最下位)</p> <div style="border: 2px solid black; padding: 10px; display: inline-block;"> <math display="block">\left[ \begin{array}{c} \text{数値定数-1} \\ \text{一意名-1} \\ \text{(算術式-1)} \end{array} \right] + \left[ \begin{array}{c} \text{数値定数-2} \\ \text{一意名-2} \\ \text{(算術式-2)} \end{array} \right]</math> </div> | 演算子の左右の引数の加算を求める。      |
| <p>図 6-8-減算演算子<br/>4 番目(最下位)</p> <div style="border: 2px solid black; padding: 10px; display: inline-block;"> <math display="block">\left[ \begin{array}{c} \text{数値定数-1} \\ \text{一意名-1} \\ \text{(算術式-1)} \end{array} \right] - \left[ \begin{array}{c} \text{数値定数-2} \\ \text{一意名-2} \\ \text{(算術式-2)} \end{array} \right]</math> </div> | 左側の引数から右側の引数を引いた値を求める。 |

COBOL 標準では、べき乗、乗算、除算、加算および減算演算子の前後に、少なくとも 1 つの空白を空ける必要がある。これによって、他の COBOL 実装との互換性を確保し、演

算子前後の空白の省略を定義する以下の特別なルールを設ける必要がなくなるため、式をコーディングするときに従うべき最適なポリシーである。

1. opensource COBOL では、べき乗、乗算、または除算の演算子の前後の空白は不要である。
2. 加算演算子の後に符号なしの数字定数が続く場合は、空白を空ける。空白を空けないと(例:「4+3」)、コンパイラは「+」を符号付き数字定数の指定として扱い、その場合、式に演算子が存在しないため「無効な式」エラーが発生する。その他では、加算演算子の前後の空白は任意となる。
3. 減算演算子の後に符号なしの数字定数が続く場合、空白を空ける。空白を空けないと(例:「4-3」)、コンパイラは「-」を符号付き数字定数の指定として扱い、その場合、式に演算子が存在しないため「無効な式」エラーが発生する。
4. どちらの引数も括弧で囲まれた式でない場合、減算演算子の前後に空白を空ける。いずれかの空白(「3-Arg」や「Arga-Argb」など)を空けなければ、コンパイラは(おそらく)存在しない定義済みの予約語やユーザ定義の名前を検索し、「「一意名」未定義」エラーを表示する。運が悪ければ、ランタイムエラーを確実に引き起こす一意名としてコンパイルされてしまうだろう。
5. 単項加算演算子の引数が、符号なしの数字定数であるとき、数字定数の一部として扱われないようにするために、単項加算演算子の後に空白を空ける必要がある(したがって、符号付き正数字定数となる)。
6. 単項否定演算子の引数が、符号なしの数字定数であるとき、数字定数の一部として扱われないようにするために、単項否定演算子の後に空白を空ける必要がある(したがって、符号付き負数字定数となる)。

ここでいくつか算術式の例を示す(説明を簡単にするため、すべての例に数字定数を使っている)。

|                                    |      |
|------------------------------------|------|
| opensource COBOL Programmers Guide | 手続き部 |
|------------------------------------|------|

| 式                        | 計算結果  | 解説                                                        |
|--------------------------|-------|-----------------------------------------------------------|
| $3 * 4 + 1$              | 13    | * は + よりも優先される。                                           |
| $2 ^ 3 * 4 - 10$         | 22    | 2 の 3 乗は 8、4 を掛けて 32、10 を引いて 22 となる。                      |
| $2 ** 3 * 4 - 10$        | 22    | 上記と同じ—opensource COBOL では「^」または「**」のいずれかを、べき乗演算子として使用できる。 |
| $3 * (4 + 1)$            | 15    | 括弧は算術式ルールを再帰的に適用し、括弧で囲まれた算術式は、他の(より複雑な)算術式の構成要素となる。       |
| $5 / 2.5 + 7 * 2 - 1.15$ | 15.35 | 整数オペランドと非整数オペランドは、自由に混在させることができる。                         |

もちろん算術式のオペランドは、数値データ項目 (DISPLAY、POINTER、または PROGRAM POINTER を除く任意の USAGE) および、数字定数をとることができる。



### 6.1.4.2. 条件式

条件式は、プログラムが実行する処理を決定する条件を識別する式であり、TRUE 値または FALSE 値を生成する。条件式は難易度の高い順に以下の 7 種類がある。

#### 6.1.4.2.1. 条件名(レベル 88 項目)

次のコードは最も単純な条件の一例である。

```
05 SHIRT-SIZE PIC 99V9.
 88 LILLIPUTIAN 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 HUMUNGOUS VALUE 19 THRU 99.9.
```

条件名「LILLIPUTIAN」、「XS」、「S」、「M」、「L」、「XL」、「XXL」、および「HUMUNGOUS」は、親データ項目(SHIRT-SIZE)内の値に基づいて、TRUE 値または FALSE 値を得る。したがって、現在の SHIRT-SIZE 値を「XL」として分類できるかどうかをテストするプログラムでは、組み合わせ条件(最も複雑なタイプの条件式)として以下のようにコード化することで、判定することができる。

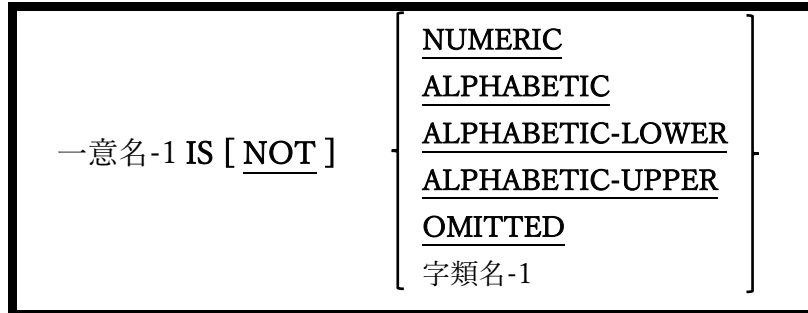
```
IF SHIRT-SIZE = 17 OR SHIRT-SIZE = 17.5
```

または次のように条件名「XL」を使用することもできる。

```
IF XL
```

## 6.1.4.2.2. 字類条件

図 6-9-字類条件構文



字類条件は、データ項目に格納されている現在のデータ型を判別する。

1. NUMERIC 字類条件では、「0」、「1」、…、「9」の文字のみが数字であると判別され、数字だけを含むデータ項目のみが IS NUMERIC クラステストを通過できる。空白、小数点、コンマ、通貨記号、プラス記号、マイナス記号、およびその他の数字以外の文字はすべて IS NUMERIC クラステストを通過できない。
2. ALPHABETIC 字類条件では、大文字、小文字、そして空白のみがアルファベットであると判別される。
3. ALPHABETIC-LOWER と ALPHABETIC-UPPER 字類条件では、空白と小文字・大文字のみクラステストを通過できる。
4. USAGE が明示的または暗黙的に DISPLAY として定義されているデータ項目のみが、NUMERIC または任意の ALPHABETIC 字類条件において使用できる。
5. 一部の COBOL 実装では、NUMERIC 字類条件での集団項目または PIC A 項目の使用、そして ALPHABETIC 字類条件での PIC 9 項目の使用は許可されていない。一方で opensource COBOL にはこのような制限はない。
6. OMITTED 字類条件は、サブルーチンが、特定の引数が引き渡されたか判別する必要がある場合に使用される。このような字類条件における一意名-1 は、サブプログラムの

「手続き部」ヘッダーの USING 句で定義された、連絡節の項目である必要がある。  
CALL からサブプログラムへの引数を省略する方法については、6.7 で説明する。

7. 字類名-1 オプションを使用すると、ユーザ定義クラスをテストできるようになる。まずは次の例のように、ユーザ定義クラス「Hexadecimal」の SPECIAL-NAME を定義する。

**SPECIAL-NAMES.**

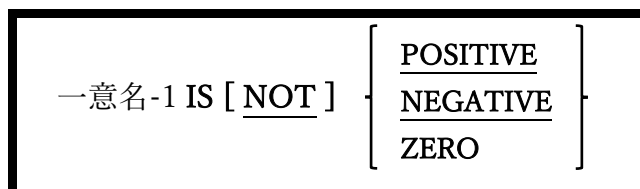
```
CLASS Hexadecimal IS '0' THRU '9', 'A' THRU 'F', 'a' THRU 'f'.
```

次は、Entered-Value に有効な 16 進数のみ入力されている場合に 150-Process-Hex-Value プロシージャを実行する、次のコードを確認する。

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

#### 6.1.4.2.3. 正負条件

図 6-10-正負条件構文

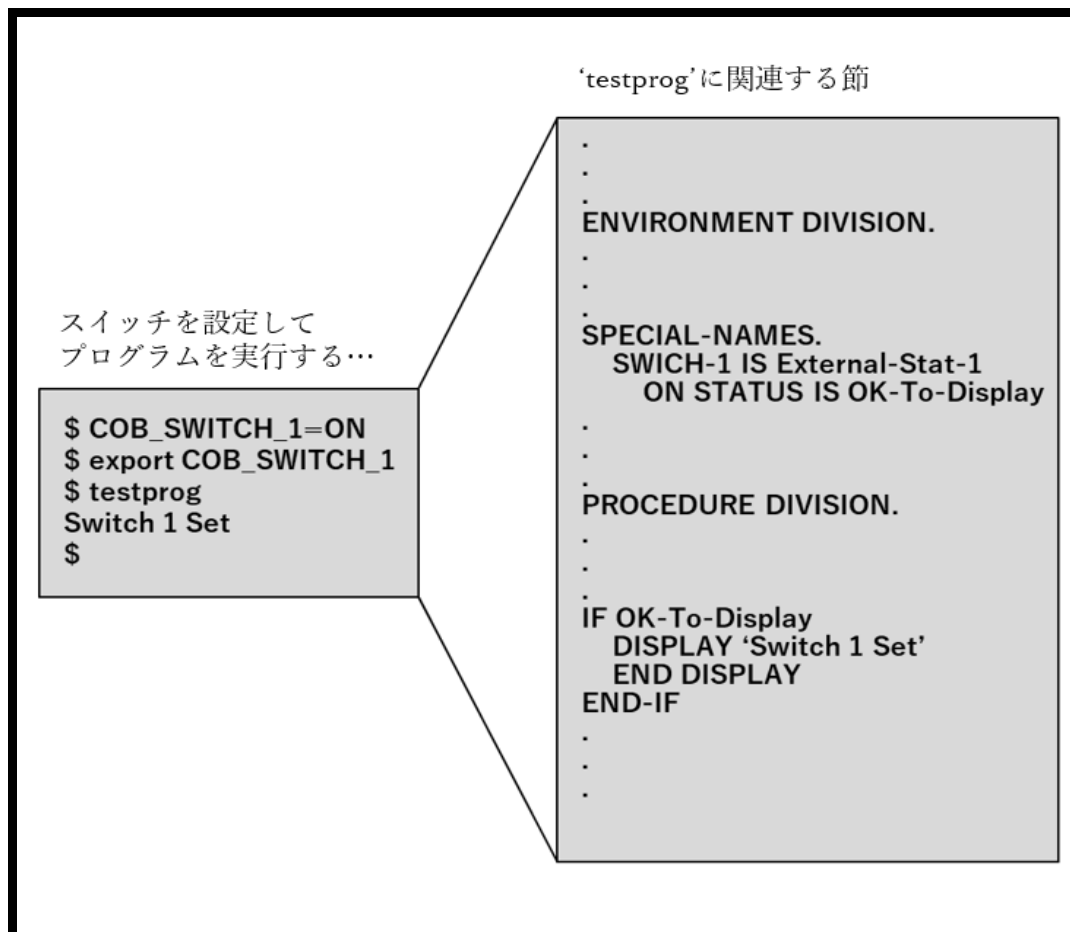


正負条件は、PIC 9 データ項目の数値状態を判別する。

- この形式の字類条件に使用できるのは、USAGE/PICTURE 句の数値として定義されたデータ項目のみである。
- POSITIVE または NEGATIVE 字類条件は一意名-1 の値がそれぞれ 0 より大きい小さい場合、ZERO 字類条件は一意名-1 の値が 0 に等しい場合、TRUE と見なす。

## 6.1.4.2.4. スイッチ状態条件

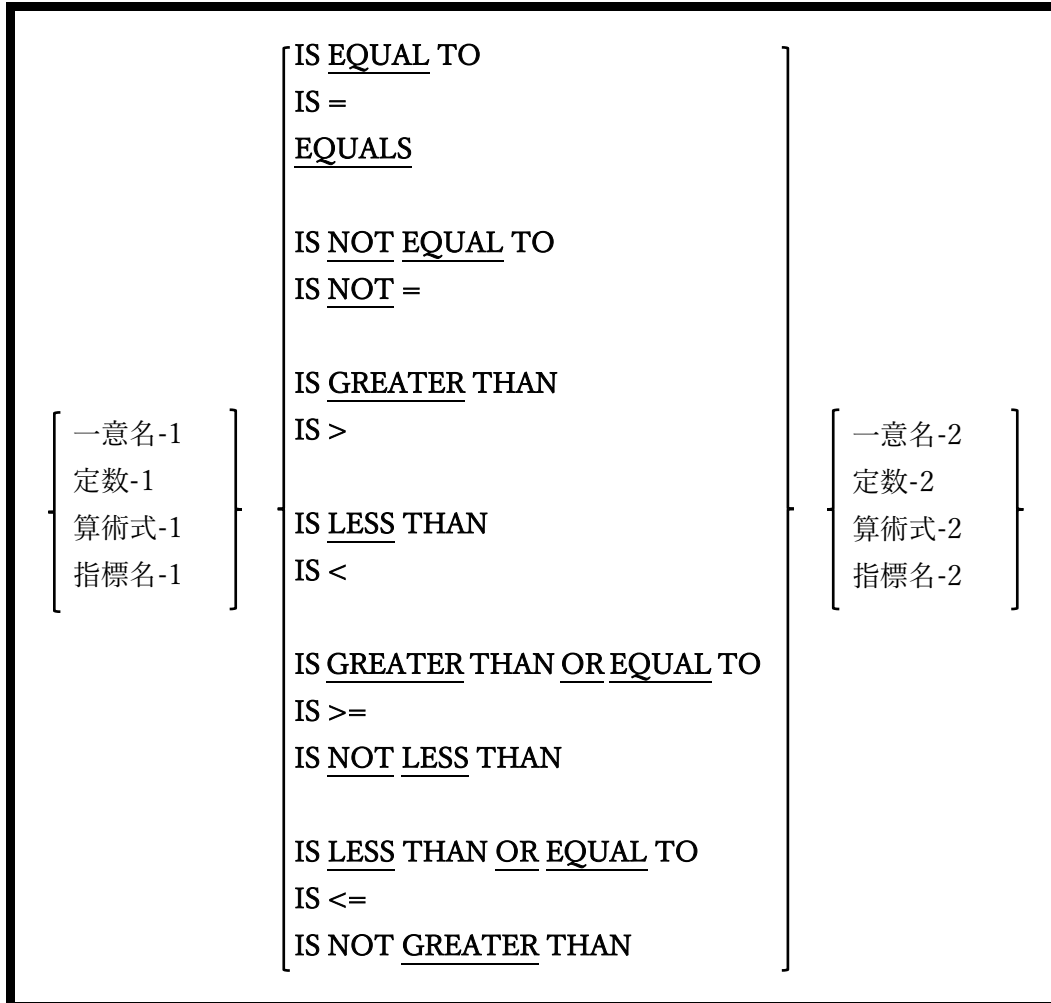
図 6-11-スイッチ状態条件



特殊名段落(4.1.4 を参照)では、外部スイッチ名を 1 つ以上の条件名と関連付けることができる。これらの条件名を使って、外部スイッチがオンまたはオフの状態にあるか判別できる。

## 6.1.4.2.5. 比較条件

図 6-12-比較条件構文



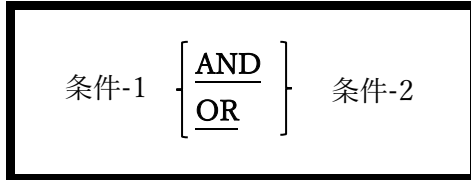
比較条件では、2つの異なる値がどのように「関連」し合っているかを判別する。

1. ある二つの数値を比較する場合、比較は実代数の値を使って実行されるため、いずれかの数値の USAGE 句と有効桁数の間に関係性はない。
2. 文字列を比較する場合、比較はプログラムの大小順序を基に行われる(4.1.2 を参照)。二つの文字列引数の長さが等しくないとき、短い方の文字列には、長い方と同じ長さになる数の空白が(右側に)埋め込まれていると見なされる。文字列の比較は、異なる文字のペアが見つかるまで、対応する文字ごとに実行される。その時点で、ペアとなった文字のそれぞれが大小順序のどこに位置するかによって、どちらがもう一方の文

字よりも大きいか(または小さいか)が決まる。

#### 6.1.4.2.6. 組み合わせ条件

図 6-13-組み合わせ条件構文



組み合わせ条件は、他の二つの条件(それ自体が組み合わせ条件の可能性があるので)によって得られた TRUE/FALSE を用いて、新たに TRUE/FALSE を判別する条件である。

1. 二つのうちいずれかの条件が TRUE の場合、OR 処理した結果は TRUE になる。二つの FALSE 条件を OR 処理した場合のみ、結果は FALSE になる。
2. AND 処理の結果を TRUE にするためには、両方の条件が TRUE である必要がある。それ以外の AND 処理の結果は全て FALSE になる。
3. 同じ演算子(OR/AND)を使って複数の類似した条件と、共通の演算子とサブジェクトを持っている左または右側の引数を繋ぐ場合、プログラムコードを省略できる。

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

以下のように省略される。

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

4. 算術式において乗算が加算よりも優先されるのと同様に、組み合わせ条件でも AND 演算子が OR 演算子より優先される。優先順位を変更する場合は、必要に応じて括弧を用いる。

|                                  |          |
|----------------------------------|----------|
| <b>FALSE OR TRUE AND TRUE</b>    | 結果：TRUE  |
| <b>(FALSE OR FALSE) AND TRUE</b> | 結果：FALSE |

FALSE OR (FALSE AND TRUE)

結果：FALSE

#### 6.1.4.2.7. 否定条件

図 6-14-否定条件構文

|        |
|--------|
| NOT 条件 |
|--------|

否定条件は NOT 演算子を用いて、条件を否定する。

1. 単項減算演算子(数値を否定する)が最も優先度の高い算術演算子であるのと同様に、NOT 演算子は論理演算子の中で、最も優先度が高い。
2. 論理演算子の既定の優先順位が望ましくないとき、条件が判別および実行される順序を明示的に示すために、括弧を用いる必要がある。

NOT TRUE AND FALSE AND NOT FALSE

FALSE AND FALSE AND TRUE

結果：FALSE

NOT (TRUE AND FALSE AND NOT FALSE)

NOT (FALSE)

結果：TRUE

NOT TRUE AND (FALSE AND NOT FALSE)

FALSE AND (FALSE AND TRUE)

結果：FALSE

#### 6.1.5. ピリオド(.)

COBOL 実装では、手続き部の完結文(センテンス)と文(ステートメント)を区別している。文とは、単一の実行可能な COBOL 命令のことである。例えば以下の例は全て文である。

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

一部の COBOL 文には「適用範囲」があり、ある文が当該文の一部であるか、関連していると考えられる。例えば以下のように、ローンの残高が 10000 ドル未満の場合は 4%、そ

れ以外は 4.5% でローンの利息が計算・表示される。

```
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
```

この例では、「IF」文の範囲内に二組の関連する文があり、それぞれ「IF」条件が TRUE の場合、または FALSE の場合に実行される。

しかし、この例には問題がある。人間がこのコードを見たとき、インデントがないことから「IF」条件が示す TRUE または FALSE の値に関係なく、DISPLAY 文が実行されると考えるだろう。残念ながら、opensource COBOL コンパイラ(またはその他の COBOL コンパイラ)にとってインデントは関係がないため、人間とは異なる識別をする。実際に、opensource COBOL コンパイラは、次のようなコードでも上記の例と同様に識別される：

```
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
```

では、DISPLAY 文が「IF」の範囲外であることを、コンパイラにどのように通知すれば良いだろうか。

そこで用いるのが完結文である。

COBOL 文は、恣意的長さの連続した文と、それに続くピリオド(.)で構成される。ピリオドは一連の文の範囲が終了することを示し、次のようにコーディングする必要がある。

```
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
```



二番目の MULTIPLY の最後にピリオドがあるのがわかるだろうか。これによって「IF」の範囲が終了し、「Loan-Balance < 10000」という式の結果に関わらず、DISPLAY が実行されるようになる。

#### 6.1.6. 動詞 / END-動詞

1985 年の COBOL 標準以前は、文の範囲が終了することを通知する唯一の方法としてピリオドが使われていた。しかし、これにはある問題があった。

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

このコードの問題は、ELSE が「IF A = 1」文ではなく、「IF B = 1」文の方に働いてしまうということだ(COBOL コンパイラはコードのインデントを判別しないことを覚えておこう)。こういった問題によって、COBOL 言語に次のような応急処置としての解決策<sup>12</sup>が追加された。

```
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".
```

---

<sup>12</sup> 例題のコードを「IF A = 1 AND B = 1」に変更すれば済む話ではあるのだが、ここでは私の主張を述べたいがために、あえて例のような表記にしている。

NEXT SENTENCE 文(6.30 参照)は、「B = 1」条件が偽の場合、次に来るピリオドの後に続く最初の文に進むよう COBOL に通知する。

1985 年の COBOL 標準と比べて、かなり優れた解決策が導入された。応急処置が必要だった COBOL 文(ステートメント)は「END-動詞」構文を用いることによって、他の文の範囲に介入することなく自らの範囲を終了させることができた。COBOL85 コンパイラであれば、以上の問題に対して次の解決策が有効だった：

```
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".
```

しかし、この新たな文法によってピリオドを用いることは時代遅れとなり、今日のセグメント分割されたプログラムは、以下のようにコーディングされている。

```
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(opensource COBOL も含む)では、手続き部の各段落に実行可能なコードがある場合、その段落には少なくとも一つの完結文が含まれている必要があるが、一般的なコーディング標準では、各段落の終わりにピリオドを一つコーディングするだけである。

COBOL 標準では、範囲符としてピリオドを使用することは変わらず有効であるため、

|                                    |      |
|------------------------------------|------|
| opensource COBOL Programmers Guide | 手続き部 |
|------------------------------------|------|

「END-動詞」の使用は任意としている。一部の文では、不要な「END-verb」範囲符が定義されている。<sup>13</sup>

既存のコードを opensource COBOL に書き込む場合は、コードが使う可能性がある言語およびコーディング標準に対応できるといった便利な機能がある。ただし、新たに opensource COBOL プログラムを作成する場合は、「END-動詞」構文を忠実に用いることを強く勧める。

### 6.1.7. 特殊レジスタ

opensource COBOL には、他の COBOL 方言と同様に、データ部で実際に定義しなくても、プログラマが自動的に使用できる多数のデータ項目が含まれている。COBOL では、レジスタや特殊レジスタなどの項目を参照する。opensource COBOL プログラムで使用できる特殊レジスタは次のとおりである。

表 6-15-特殊レジスタ

| レジスタ名          | 暗黙の COBOL<br>PIC/USAGE 句 <sup>14</sup> | 使用方法                                                                                                                                                                                                                                      |
|----------------|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LINAGE-COUNTER | BINARY-LONG<br>SIGNED                  | このレジスタのオカレンスは、LINAGE 句を持つ SELECT で指定された各ファイルに存在する(5.1 を参照)。FD に LINAGE 句があるファイルが複数ある場合、このレジスタへの明示的な参照には修飾が必要である(「OF ファイル名」を使用)。<br><br>このレジスタの値は、ページ本体内の現在の論理行番号になる(LINAGE 句が論理ページを構成する方法については 5.1 を参照)。<br><br><b>このレジスタの内容は変更してはいけな</b> |

<sup>13</sup> 例えば STRING(6.43)と UNSTRING(6.49)には、範囲符が必要なステートメントにオプションを導入するといった将来的な標準に向けての計画はあるのだろうか？

<sup>14</sup> PICTURE 句または USAGE 句の仕様の説明については 5.3 を参照。

|                                    |      |
|------------------------------------|------|
| opensource COBOL Programmers Guide | 手続き部 |
|------------------------------------|------|

|                           |                    |                                                                                                                                                                                                                                                                           |
|---------------------------|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                           |                    | い。                                                                                                                                                                                                                                                                        |
| NUMBER-OF-CALL-PARAMETERS | BINARY-LONG SIGNED | <p>このレジスタには、サブプログラムに渡される引数の数が含まれている。メインプログラムで参照されると、その値はゼロになる。</p> <p>同じデータを取得する別の方法については、7.3.1.7のC\$NARG組み込みサブルーチンのドキュメントを参照。</p>                                                                                                                                        |
| RETURN-CODE               | BINARY-LONG SIGNED | <p>このレジスタは、数値データ項目を提供する。サブルーチンは、それ CALL したプログラムに制御を戻す前に値を MOVE したり、メインプログラムがオペレーティングシステムに制御を返す前に値を MOVE したりすることができる。</p> <p>ほとんどの組み込みサブルーチン(7.3)が、このレジスタを使用して値を返す。</p> <p>これらの値は一規則により—RETURN-CODE 値を設定したプログラムが実行しようとしていたプロセスの成功(通常は値 0)または失敗(通常は 0 以外の値)を示すために使用される。</p> |
| SORT-RETURN               | BINARY-LONG SIGNED | <p>このレジスタは、RELEASE 文または RETURN 文の成功または失敗のステータスを示すために使用される。成功の場合は値 0 が返り、値 16 が返ってきた場合は失敗を示す。RETURN 文の「AT END」状態は、失敗とは見なされない。</p>                                                                                                                                          |
| WHEN-COMPILED             | See “Usage”        | <p>このレジスタには、プログラムがコンパイルされた日時が「mm/dd/yyhh.mm.ss」の形式で含まれている。返ってくるのは 2 桁の年のみであることに注意すること。</p>                                                                                                                                                                                |

### 6.1.8. ファイルへの同時アクセス制御

データファイルの操作は、COBOL 言語の大きな強みの 1 つである。複数のプログラムが同じファイルに同時にアクセスしようとする可能性を対処するため、COBOL 言語に組み込まれている機能がある。複数プログラムの同時アクセスは、ファイル共有とレコードロックの 2 つの方法で処理される。

すべての opensource COBOL 実装がファイル共有およびレコードロックオプションをサポートしているわけではない。それらが構築されたオペレーティングシステムと、特定の opensource COBOL 実装が生成されたときに使用されたビルドオプションによって異なる。

#### 6.1.8.1. ファイル共有

opensource COBOL は、プログラムがファイルを開こうとしたときに適用されるファイル共有の概念によって、最水準でファイルの同時アクセスを制御する(6.31 を参照)。これは「`fcntl()`」と呼ばれる UNIX オペレーティングシステムルーチンを介して実行される。そのモジュールは現在 Windows でサポートされておらず<sup>15</sup>、MinGW Unix エミュレーションパッケージに含まれていない。MinGW 環境を使用して作成された opensource COBOL ビルドは、ファイル共有制御をサポートできなくなる—そのような環境ではファイルが常に共有される。Windows で Cygwin 環境を使用して作成された opensource COBOL ビルドは、「`fcntl()`」にアクセスできると思われるため、ファイル共有をサポートするだろう。もちろん、opensource COBOL の Unix ビルドや MacOS ビルドは<sup>16</sup>、「`fcntl()`」が Unix に組み込まれているため、BDB を使用しても問題はない。

---

<sup>15</sup> Windows には「`fcntl()`」と同様の機能があるが、BDB パッケージはそれらの機能を利用するようにコーディングされていない。UNIX と Windows の両方の同時アクセスルーチン(VBISAM など)をサポートする高度なファイル I/O パッケージの使用は、現在、著者によって調査中である。

<sup>16</sup> Apple Computer の MacOS X オペレーティングシステムは UNIX のオープンソースバージョンに基づいているため、「`fcntl()`」のサポートが含まれている。

|                                    |      |
|------------------------------------|------|
| opensource COBOL Programmers Guide | 手続き部 |
|------------------------------------|------|

OPEN の成功に課せられる制限は、プログラムがファイルを CLOSE するか、終了するまで残る。

ファイルへの同時アクセスをファイルレベルで制御するには、次の 3 つの方法がある。

| 共有<br>オプション  | 効果                                                                                                                                                                |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ALL<br>OTHER | あなたのプログラムがファイルを開いた後に、他のプログラムがファイルを開こうとしても制限されない。これは SHARING 句が指定されなかったときの既定値である。                                                                                  |
| NO<br>OTHER  | あなたのプログラムがファイルを使用している限り、他の <u>どんな</u> プログラムによる、 <u>どんな</u> ファイルアクセスも許可しない。他のプログラムによって行われた OPEN の試行は、あなたがファイルを閉じるまでファイル状態コード 37(「ファイルアクセスが拒否されました」)で失敗する(6.9 を参照)。 |
| READ<br>ONLY | あなたがファイルを開いている間、他のプログラムが INPUT のためにファイルを開くことを許可する。他の目的で OPEN を試行すると、ファイル状態コード 37 で失敗する。                                                                           |

誰かが最初にファイルにアクセスし、ファイル共有を制限する共有オプションでファイルを OPEN した場合、当然あなたのプログラムはアクセスに失敗する。

#### 6.1.8.2. レコードロック

レコードロックは、ファイル(通常は ORGANIZATION INDEXED ファイル)にアクセスするための単一の制御ポイントを提供する高度なファイル管理ソフトウェアによってサポートされている。レコードロックを実行できるランタイムパッケージの 1 つは、Berkely DB(BDB)パッケージである。様々な I/O 文は一他の同時実行プログラムによる一アクセスしたばかりのファイルレコードへのアクセスに制限を課することができる。これらの制限は、レコードにロックをかけることによって構文的に課せられる。OPEN 時に課せられたファイル共有の制限がファイル全体へのアクセスを妨げなかったと仮定すると、ファイル内の他のレコードは引き続き利用可能である。

|                                    |      |
|------------------------------------|------|
| opensource COBOL Programmers Guide | 手続き部 |
|------------------------------------|------|

ロックを保持しているプログラムが終了するか、ファイルに対して CLOSE 文(6.9)、UNLOCK 文(6.48)、COMMIT 文(6.10)、または ROLLBACK 文(6.37)を実行するまでロックが有効である。

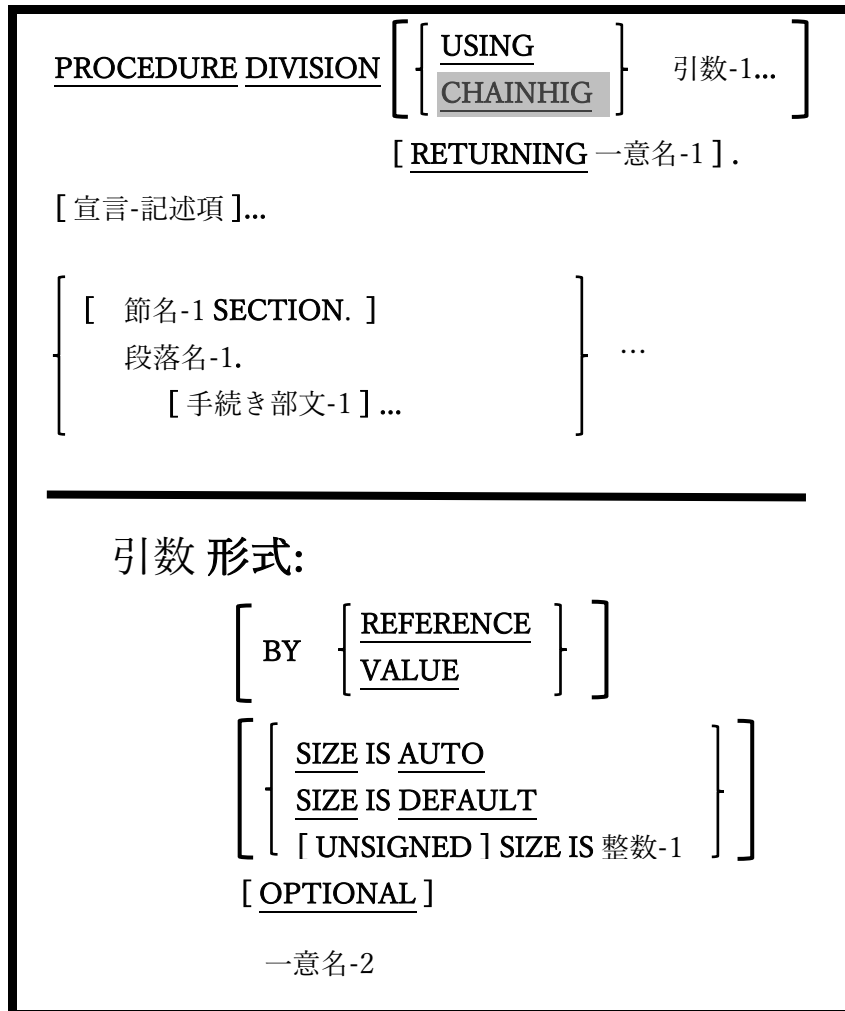
レコードロックオプション(すべてのオプションがすべての文で利用できるとは限らない)を次の表で示している。

| レコードロック<br>オプション                        | 効果                                                                                                                                                                                                                                                                                                         |
|-----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| WITH LOCK                               | 他のプログラムによるレコードへのアクセスは拒否される。                                                                                                                                                                                                                                                                                |
| WITH NO<br>LOCK                         | レコードはロックされない。すべての文で有効なロックオプションが指定されなかったときの既定値である。                                                                                                                                                                                                                                                          |
| IGNORING<br>LOCK<br>WITH IGNORE<br>LOCK | レコードを読み取る場合にのみ有効なオプション—他のプログラムによって保持されているロックは無視するよう opensource COBOL に通知する。<br>左に示した 2 つのオプションは同義である。                                                                                                                                                                                                      |
| WITH WAIT                               | レコードを読み取る場合にのみ有効なオプション—読み取るレコードに保持されているロックが解放されるのをプログラムが待機していることを opensource COBOL に通知する。<br>このオプションがないと、ロックされたレコードの読み取りはすぐに中止され、ファイル状態コード 47 が返される。<br>このオプションを使用すると、プログラムは事前に設定された時間だけロックが解放されるのを待機する。事前に設定された待機時間内にロックが解除されると、読み取りは成功する。ロックが解除される前に事前に設定された待機時間が経過すると、読み取りの試行は中止され、ファイル状態コード 47 が発行される。 |

使用している opensource COBOL ビルドが BDB を利用するように構成されている場合、実行時環境変数 DB\_HOME を使って(7.2.4 を参照)レコードロックを使用できる。

## 6.2. 記述形式

図 6-16-記述形式構文



手続き部の最初の(オプション)セグメントは、「宣言」と呼ばれる特別な領域となっている。この領域内では、特定のイベントが発生した場合のみ実行される特殊な「トラップ」としての処理ルーチンを定義できる。これについては次の 6.3 で説明する。

手続き型および論理型プログラムが書かれている節や段落は「宣言」に従う。手続き部は独自の節や段落を作成できる COBOL 部門の一つである。

1. USING 句と RETURNING 句は、サブルーチンとして機能しているプログラムへの引数を定義する。これらの句によって指定されたすべての一意名は、USING 句および、

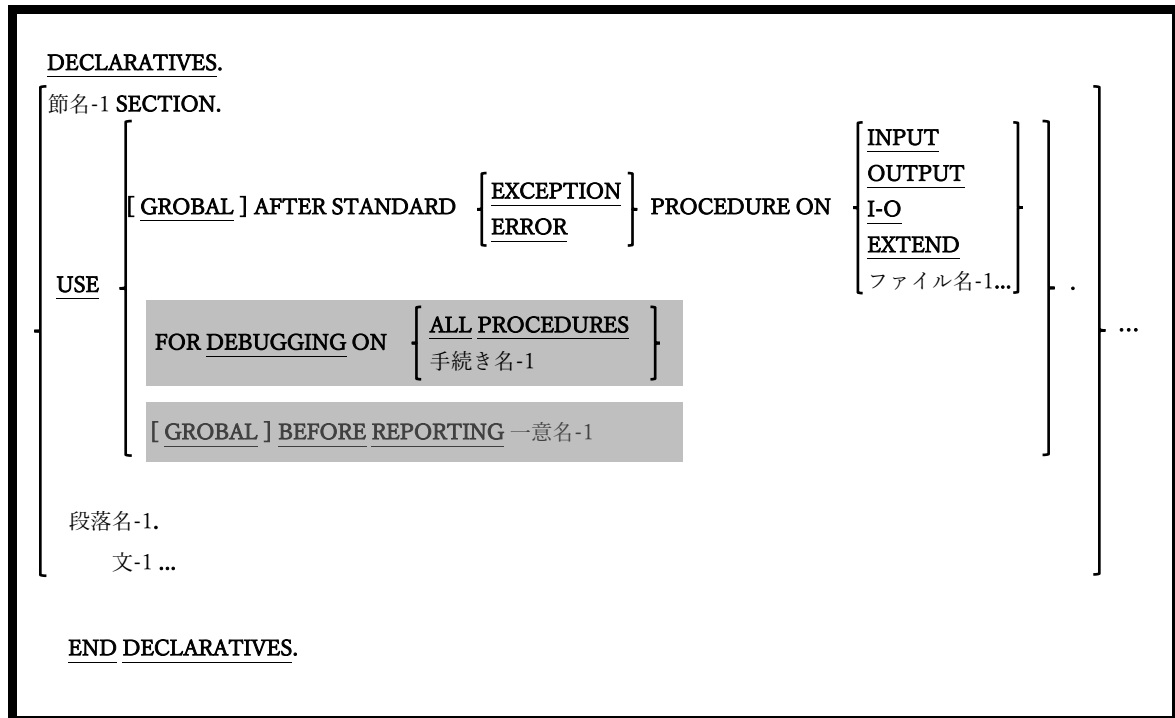


または RETURNING 句が表示されるプログラムの連絡節で定義する必要がある。

2. CHAINING 句は、CHAIN 文を介した他のプログラムによって呼び出されるプログラム内でのみ使うことができる。CHAINING 句で指定された一意名は、CHAINING 句が表示されるプログラムの連絡節で定義する必要がある。この CHAINING 句は opensource COBOL においては構文的に使用可能となつてはいるが、それ以外では機能しないため、CHAIN 文を使おうとした場合は拒否される。
3. ユーザ定義関数(現在 opensource COBOL では使用不可)での使用を目的としているが、RETURNING 句は、値が返されるサブプログラムへの引数を指定し、それを文書化する手段として用いることができる。
4. BY REFERENCE 句は、プログラムの引数に対応するデータ項目のアドレスがプログラムに渡されることを示す。このプログラムでは、BY REFERENCE 引数の内容を変更することができるが、BY REFERENCE は、すべての USING/CHAINING 引数において、BY REFERENCE、BY VALUE が指定されなかったときの既定値である(ここで CHAINING 引数は必ず BY REFERENCE でなければならない)。
5. BY VALUE 句では、引数に対応する呼び出し側プログラムからのデータ項目の読み込み専用コピーがプログラムに引き渡される。BY VALUE 引数の内容は、サブプログラムによって変更することはできない。
6. USING 句のメカニズムは、COBOL の一部のメインフレーム実装の場合と同様に、opensource COBOL プログラムがコマンドライン引数を取得することではない。プログラムのコマンドライン引数取得方法については、この後記述する ACCEPT 文が参考になる。
7. SIZE 句は、引き渡された引数のサイズ(バイト単位)を指定し、SIZE IS AUTO 句(既定値)では、呼び出し側プログラムの項目サイズに基づいて、引数のサイズが自動で決定される。残りの SIZE オプションでは、特定のサイズを強制的に決定でき、SIZE IS DEFAULT は、UNSIGNED(符号なし) SIZE IS 4 と同様のサイズを示す。

### 6.3. 宣言の記述形式

図 6-17- 宣言構文



プログラマは手続き部の宣言領域内で、プログラム実行時に発生する可能性のある特定のイベントを遮断する、一連の「トラップ」ルーチンを定義することができる。

1. RWCS は現在 opensource COBOL においてサポートされていないため、USE BEFORE REPORTING 句は構文的には認識されても拒否される。
2. USE FOR DEBUGGING 句も同様に、構文的に認識されても無視されてしまう。「-Wall」または「-W」のコンパイラスイッチを使用すると、この機能がまだ実装されていないことを示す警告メッセージが表示される。
3. USE AFTER STANDARD ERROR PROCEDURE 句では、指定された I/O タイプで（または指定されたファイルに対して）障害が発生したときに呼び出されるルーチンを定義する。
4. GLOBAL オプションを使用すると、同じコンパイル単位内のすべてのプログラムにお

いて宣言型プロシージャを使用できる。

5. 宣言ルーチン(任意の型)は、PERFORM 文を介して参照する場合を除いて、宣言範囲外のプロシージャを参照することはできない。

## 6.4. ACCEPT

### 6.4.1. ACCEPT 文の書き方 1 — コンソールからの読み取り

図 6-18-ACCEPT 構文(コンソールからの読み取り)

```
ACCEPT 一意名
 [FROM ニーモニック名]
 [END-ACCEPT]
```

コンソールウィンドウから値を読み取り、それをデータ項目(一意名)に格納するために使用する。

1. FROM 句を使う場合、指定するニーモニック名は SYSIN または CONSOLE のいずれかであるか、または、特殊名段落を介してこれら 2 つのいずれかに割り当てられたユーザ定義のニーモニック名である必要がある。SYSIN と CONSOLE は同じ意味を持つものとして使われ、どちらもコンソールウィンドウを参照する。
2. FROM 句が指定されていない場合は、FROM CONSOLE が指定されたとみなす。

### 6.4.2. ACCEPT 文の書き方 2 — コマンドライン引数の取得

図 6-19-ACCEPT 構文(コマンドライン引数)

```
ACCEPT 一意名
 FROM { COMMAND-LINE
 ARGUMENT-NUMBER
 ARGUMENT-VALUE [例外処理] }
 [END-ACCEPT]
```

プログラムのコマンドラインから引数を取得するために使用する。

1. COMMAND-LINE オプションから受け取ると、プログラムを実行したコマンドライ

ンで入力された全ての引数を、指定した通りに取得できるが、返ってきたデータを意味のある情報に解析する必要がある。

2. ARGUMENT-NUMBER から受け取る場合、コマンドラインから引数を解析し、発見した引数の数を返すように opensource COBOL ランタイムシステムに要求する。解析は、次のようにオペレーティングシステムのルールに従って実行される。

- 引数は、文字間の空白を引数間の区切り文字として扱うことで区切られる。2つの空白以外の値を区切る空白の数とは無関係である。
- 二重引用符(“)で囲まれた文字列は、引用符内に埋め込まれる可能性のある空白の数(空白が存在する場合は)に関係なく、単体の引数として扱われる。
- Windows システムでは、一重引用符またはアポストロフィ文字(‘)は、他のデータ文字と同じように扱われ、文字列を示すことはできない。

3. ARGUMENT-VALUE から受け取る場合、コマンドラインから引数を解析し、現在の ARGUMENT-NUMBER レジスタにある引数を返すように opensource COBOL ランタイムシステムに要求する<sup>17</sup>。解析は、上記の 2 項で記載したルールに従って実行される。

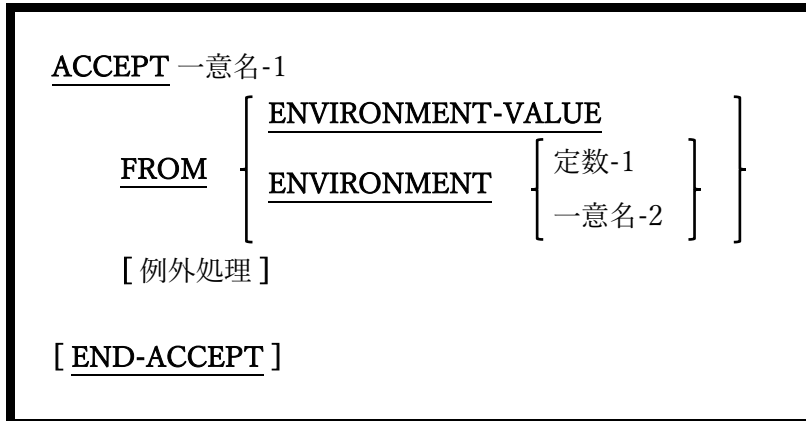
4. オプションの例外処理の構文と用法については、6.4.7 で説明する。

---

<sup>17</sup> DISPLAY 文の書き方 2 を使って、ARGUMENT-NUMBER を目的の値に設定する。

### 6.4.3. ACCEPT 文の書き方 3 — 環境変数値の取得

図 6-20-ACCEPT 構文(環境変数値の取得)



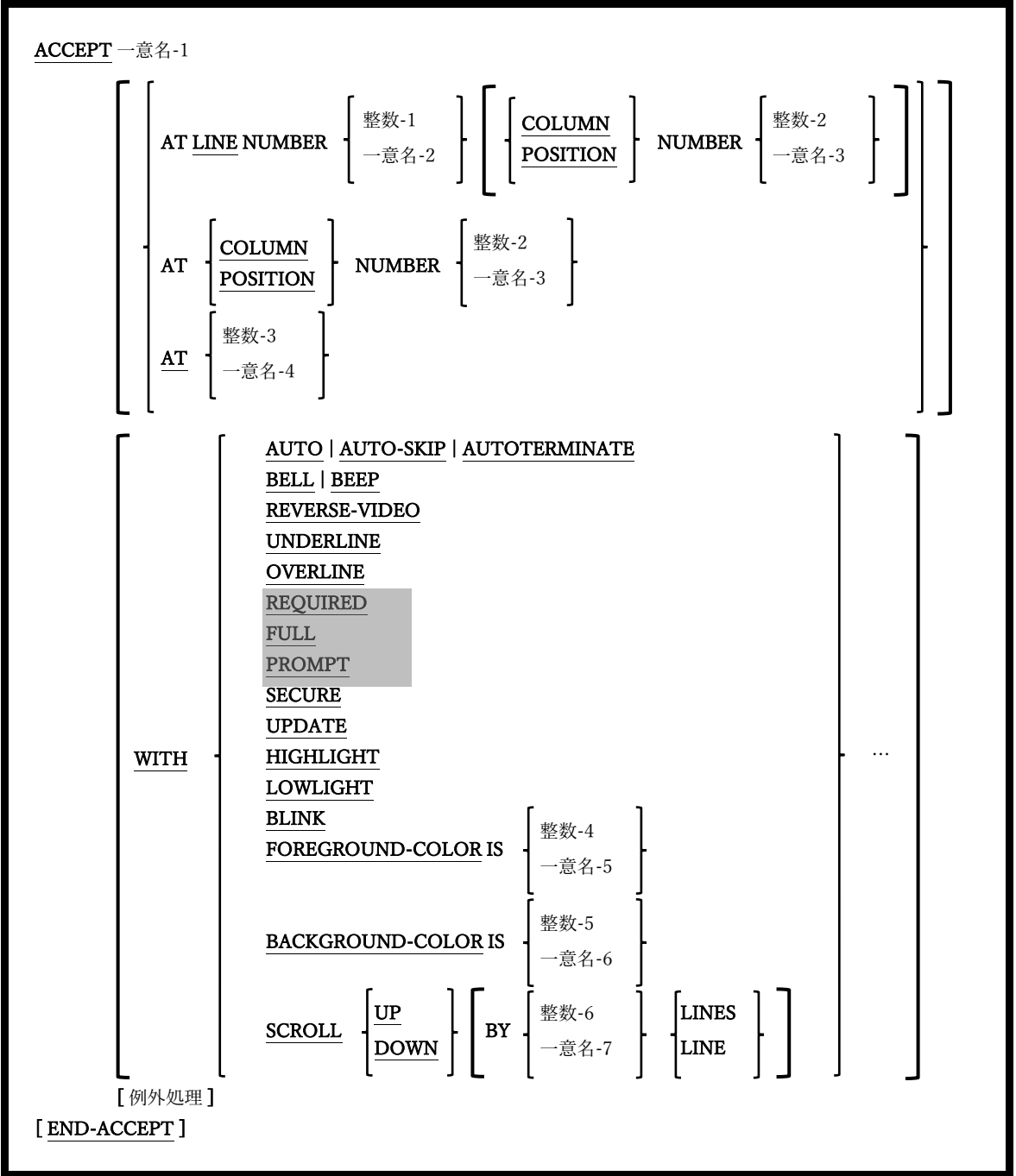
プログラムのコマンドラインから引数を取得するために使用する。

1. ENVIRONMENT-VALUE から受け取る場合、現在の ENVIRONMENT-NAME レジスタにある環境変数の値を取得するように opensource COBOL ランタイムシステムに要求する<sup>18</sup>。
2. 環境変数値を取得する、より簡単なアプローチは「ACCEPT … FROM ENVIRONMENT」を使うことである。その書き方では、ACCEPT コマンド自体で取得する環境変数を指定する。
3. オプションの例外処理の構文と使用法については、6.4.7 で説明する。

<sup>18</sup> DISPLAY 文の書き方 3 を使って ENVIRONMENT-NAME を目的の環境変数名に設定する。

#### 6.4.4. ACCEPT 文の書き方 4 — 画面データの取得

図 6-21-ACCEPT 構文(画面データの取得)



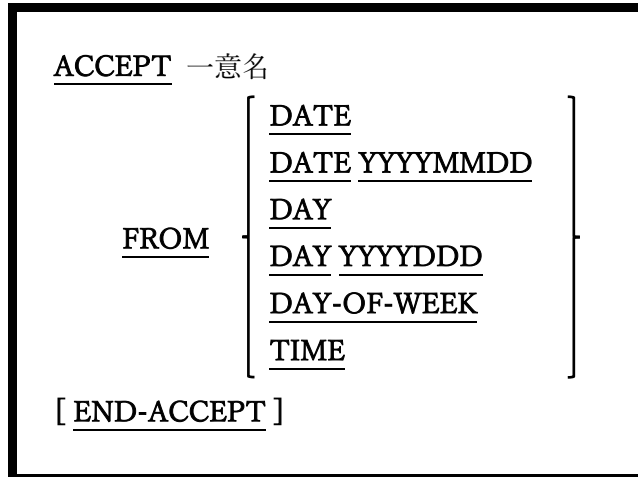
画面節で定義されたデータ項目を利用して、形式化されたコンソールウィンドウ画面からデータを取得するために使用する。

1. 一意名-1 が SCREEN SECTION で定義されている場合、すべてのカーソル位置(AT)および属性指定(WITH)は SCREEN SECTION 定義から取得され、ACCEPT で指定されたものはすべて無視される。AT および WITH オプションは、SCREEN SECTION で定義されていないデータ項目を受け入れる場合にのみ使う。
2. AT 句は、画面が読み取られる前に、カーソルを画面上の特定の場所に配置する手段を提供する。定数-3/一意名-4 の値は 4 桁である必要があり、最初の 2 桁はカーソルを配置する行、最後の 2 桁は列を示す。
3. UPDATE と SCROLL を除いて、ほとんどの WITH オプションについて 5.6 で説明している。SCROLL 以外の WITH オプションは、1 回だけ指定する必要がある。
4. UPDATE オプションは、新しい値を受け取る前に一意名-1 の現在の内容を表示する句である。
5. SCROLL オプションを使用すると、画面に値が表示される前に、画面上の内容の全体が指定された行数だけ上下にスクロールされる。SCROLL UP 句や SCROLL DOWN 句を指定することもできる。LINES 指定がない場合は「1 LINE」と見なされる。
6. オプションの例外処理の構文と使用法については、6.4.7 で説明する。



#### 6.4.5. ACCEPT 文の書き方 5 — 日付/時刻の取得

図 6-22-ACCEPT 構文(日付/時刻の取得)構文



システムの現在の日付や時刻を取得してデータ項目に保存するために使用する。

1. システムから取得したデータ、および構造化された書き方は、次の表のように異なっている。

表 6-23-ACCEPT オプション(日付/時刻の取得)

| オプション         | 取得データ   | 一意名-1 の書き方                                                                                        |
|---------------|---------|---------------------------------------------------------------------------------------------------|
| DATE          | 西暦表示の日付 | 01 CURRENT-DATE.<br>05 CD-YEAR PIC 9(2).<br>05 CD-MONTH PIC 9(2).<br>05 CD-DAY-OF-MONTH PIC 9(2). |
| DATE YYYYMMDD | 西暦表示の日付 | 01 CURRENT-DATE.<br>05 CD-YEAR PIC 9(4).<br>05 CD-MONTH PIC 9(2).<br>05 CD-DAY-OF-MONTH PIC 9(2). |
| DAY           | 和暦表示の日付 | 01 CURRENT-DATE.<br>05 CD-YEAR PIC 9(2).<br>05 CD-DAY-OF-YEAR PIC 9(3).                           |
| DAY YYYYDDD   | 和暦表示の日付 | 01 CURRENT-DATE.<br>05 CD-YEAR PIC 9(4).<br>05 CD-DAY-OF-YEAR PIC 9(3).                           |
| DAY-OF-WEEK   | 曜日      | 01 CURRENT-DATE.<br>05 CD-DAY-OF-WEEK PIC 9(1).<br>88 MONDAY VALUE 1.<br>88 TUESDAY VALUE 2.      |

|                                    |      |
|------------------------------------|------|
| opensource COBOL Programmers Guide | 手続き部 |
|------------------------------------|------|

|      |      |                                                                                                                                       |
|------|------|---------------------------------------------------------------------------------------------------------------------------------------|
|      |      | 88 WEDNESDAY VALUE 3.<br>88 THURSDAY VALUE 4.<br>88 FRIDAY VALUE 5.<br>88 SATURDAY VALUE 6.<br>88 SUNDAY VALUE 7.                     |
| TIME | 現在時刻 | 01 CURRENT-TIME.<br>05 CT-HOURS PIC 9(2).<br>05 CT-MINUTES PIC 9(2).<br>05 CT-SECONDS PIC 9(2).<br>06 CT-HUNDREDTHS-OF-SECS PIC 9(2). |

#### 6.4.6. ACCEPT 文の書き方 6 — 画面サイズデータの取得

図 6-24-ACCEPT(画面サイズデータの取得)構文

```
ACCEPT 一意名
 FROM { LINES
 COLUMNS }
[END-ACCEPT]
```

プログラムが実行されているコンソールウィンドウの(文字位置での)表示可能なサイズを取得するために使用する。

1. Windows コンソールウィンドウなど、ウィンドウの論理サイズが物理コンソールウィンドウの論理サイズをはるかに超える可能性のある環境では、物理コンソールウィンドウのサイズを取得する。

#### 6.4.7. ACCEPT 文の例外処理

図 6-25-ACCEPT 例外処理構文

```
[ON EXCEPTION
 命令文-1]
[NOT ON EXCEPTION
 命令文-2]
```

ACCEPT 文の一部の書き方において EXCEPTION 句と NOTEXCEPTION 句が利用可能で、ACCEPT 文の失敗または成功時に実行されるコードを(それぞれ)指定できる。ACCEPT 文ではリターンコードまたはステータスフラグを設定しないため、これが成功と失敗を検出する唯一の方法となる。

## 6.5. ADD

### 6.5.1. ADD 文の書き方 1 — ADD TO

図 6-26-ADD 構文(GIVING)

```
ADD [LENGTH OF] [定数-1
一意名-1] ...
TO { 一意名-2 [ROUNDED] } ...
[ON SIZE ERROR 命令文-1]
[NOT ON SIZE ERROR 命令文-2]
[END-ADD]
```

TO の前にあるすべての引数(一意名-1 または定数-1)の算術和を生成し、その合計値を TO の後にリストされている各一意名(一意名-2)に追加する。

1. 一意名-1 および一意名-2 は、編集不可の数値データ項目でなければならない。
2. 定数-1 は数値定数でなければならない。
3. 整数以外の結果が生成されるか、あるいは ROUNDED キーワードを持つ一意名-2 データ項目に割り当てられた場合、一意名-2 に格納された結果は、数学的規則に従って最下位桁を切り上げられる。例えば、PICTURE が 99V99 で、格納される結果が 12.152 の場合、値は 12.15 になるが、結果が 76.165 の場合では 76.17 の値が格納される。
4. LENGTH OF 句が定数-1 または一意名-1 で使用されている場合、計算プロセスの中で使われる算術値は、データ項目または定数のバイト単位での長さであり、実際の値ではない。
5. ON SIZE ERROR 句を使うと、一意名-2 の項目に格納される結果がその項目の容量を超えた場合に実行されるコードを指定することができる。例えば、PICTURE が 99V99 で、格納される結果が 101.43 の場合、SIZE ERROR 条件が発生する。ON SIZE

ERROR 句がない場合、opensource COBOL は 01.43 の値を項目に格納する。ON SIZE ERROR 句を使用すると、一意名-2 項目の値は変更されずに、命令文-1 が実行される。例として、デモプログラムとその出力を示した(図 6-27)。また、「EXCEPTION」組み込み関数についても説明している(6.1.7 参照)。

図 6-27-ON SIZE ERROR 句を使用するサンプルプログラム

```
1. IDENTIFICATION DIVISION.
2. PROGRAM-ID. corrdemo.
3. DATA DIVISION.
4. WORKING-STORAGE SECTION.
5. 01 Item-1 VALUE 1 PIC 99V99.
6. PROCEDURE DIVISION.
7. 100-Main SECTION.
8. P1.
9. ADD 19 81.43 TO Item-1
10. ON SIZE ERROR
11. DISPLAY 'Item-1:' Item-1
12. DISPLAY 'Error: ' FUNCTION EXCEPTION-STATUS
13. DISPLAY 'where: ' FUNCTION EXCEPTION-LOCATION
14. DISPLAY ' what: ' FUNCTION EXCEPTION-STATEMENT
15. END-ADD.
16. STOP RUN.
```

When executed, the program produces the following output:

```
Item-1:0100
Error: EC-SIZE-OVERFLOW
Where: corrdemo; P1 OF 100-Main; 9
What: ADD
```

6. NOT ON SIZE ERROR 句を指定すると、ADD 文で項目サイズのオーバーフロー条件が発生しなかった場合に命令文が実行される。

## 6.5.2. ADD 文の書き方 2 — ADD GIVING

図 6-28-ADD 構文(GIVING)

```
ADD { [LENGTH OF] { 定数-1
 一意名-1 } } ...
 [TO 一意名-2]
 GIVING { 一意名-3 [ROUNDED] } ...
 [ON SIZE ERROR 命令文-1]
 [NOT ON SIZE ERROR 命令文-2]
 [END-ADD]
```

TO の前にあるすべての引数(一意名-1 または定数-1)の算術和を生成し、一意名-2(存在する場合)に合計値を追加、GIVING の後にリストされている一意名(一意名-3)の内容を合計値に置き換える。

1. 一意名-1 および一意名-2 は、編集不可の数値データ項目でなければならない。
2. 一意名-3 は数値データ項目でなければならないが、編集可能な場合もある。
3. 定数-1 は数字定数でなければならない。
4. 一意名-2 の内容は変更できない。
5. ROUNDED、LENGTH OF、ON SIZEERROR および NOTON SIZE ERROR 句の使い方と動作は、6.5.1 ADD 文の書き方 1 で説明している。

### 6.5.3. ADD 文の書き方 3 — ADD CORRESPONDING

図 6-29-ADD 構文(CORRESPONDING)

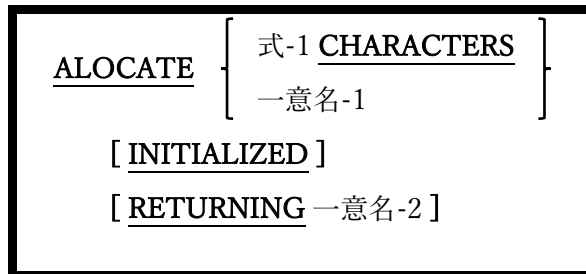
```
ADD CORRESPONDING 一意名-1 TO 一意名-2 [ROUNDED]
 [ON SIZE ERROR 命令文-1]
 [NOT ON SIZE ERROR 命令文-2]
 [END-ADD]
```

二つの一意名に従属して見つかったデータ項目に対応する個々の ADD TO 文と、同等のコードを生成する。

1. 対応するものを識別するための規則については、6.28.2 – MOVE CORRESPONDING で説明している。
2. ROUNDED、ON SIZEERROR および NOT ON SIZE ERROR 句の使い方と動作は、6.5.1 ADD 文の書き方 1 で説明している。

## 6.6. ALLOCATE

図 6-30-ALLOCATE 構文



ALLOCATE 文は、実行時に動的にメモリを割り当てるために使用する。

1. 式-1 を使う場合、ゼロ以外の正の整数値を持つ算術式である必要がある。「式-1 CHARACTERS」オプションを使う時は、06FEB2009 バージョンの構文パーサーを混乱させないように式を括弧で囲んで、「一意名-1」オプションと間違えないように気を付ける。パーサーが「混乱」する可能性については、今後、opensource COBOL 1.1 tarball で修正される予定である。
2. 一意名-1 は、WORKING-STORAGE または LOCAL STORAGE の BASED 属性で定義された 01 レベル項目である必要がある。連絡節で定義されている 01 項目にすることもできるが推奨しない。
3. 一意名-2 は USAGE POINTER データ項目である必要がある。
4. RETURNING 句は、割り当てられたメモリブロックのアドレスを、指定された USAGE POINTER 項目に返す。その USAGE POINTER 項目に対して FREE 文(6.19)が発生した場合に備え、opensource COBOL は割り当てられたメモリブロックが最初に要求されたサイズの情報を保持している。
5. 「一意名-1」オプションを使うと、INITIALIZE は一意名-1 の定義に存在する PICTURE 句および VALUE 句(存在する場合)に従って、割り当てられたメモリブロックを初期化する。INITIALIZE 文については、6.24 で説明している。



6. 「式-1CHARACTERS」オプションでは、INITIALIZE は割り当てられたメモリブロックをバイナリゼロに初期化する。
7. INITIALIZE 句を使わない場合、割り当てられたメモリの初期内容は、プログラムが実行されているオペレーティングシステムに対して有効なメモリ割り当てのルールに委ねられる。
8. 基本的な使用法は二つあり、最も単純なものは次の例である。

**ALLOCATE My-01-Item**

**My-01-Item** の定義済みサイズ(BASED 属性で定義されている必要がある)と同じサイズのストレージブロックが割り当てられる。この時ストレージブロックのアドレスが **My-01-Item** の基本アドレスとなり、そのブロックと下位データ項目がプログラム内で使用できるようになる。

二つ目の使用法は以下の通りである。

**ALLOCATE LENGTH OF My-01-Item CHARACTERS RETURNING The-Pointer.  
SET ADDRESS OF My-01-Item TO The-Pointer.**

ALLOCATE 文は、**My-01-Item** に必要な分と全く同じサイズのメモリブロックを割り当て、アドレスはポインタ変数に返される。次に SET 分は、**My 01-Item** のアドレスを「ベース」として、ALLOCATE によって作成されたメモリブロックのアドレスにする。

上記二つの使用法の唯一の機能上の違いとしては、最初の例で、INITIALIZED 句がある場合は尊重されることである。

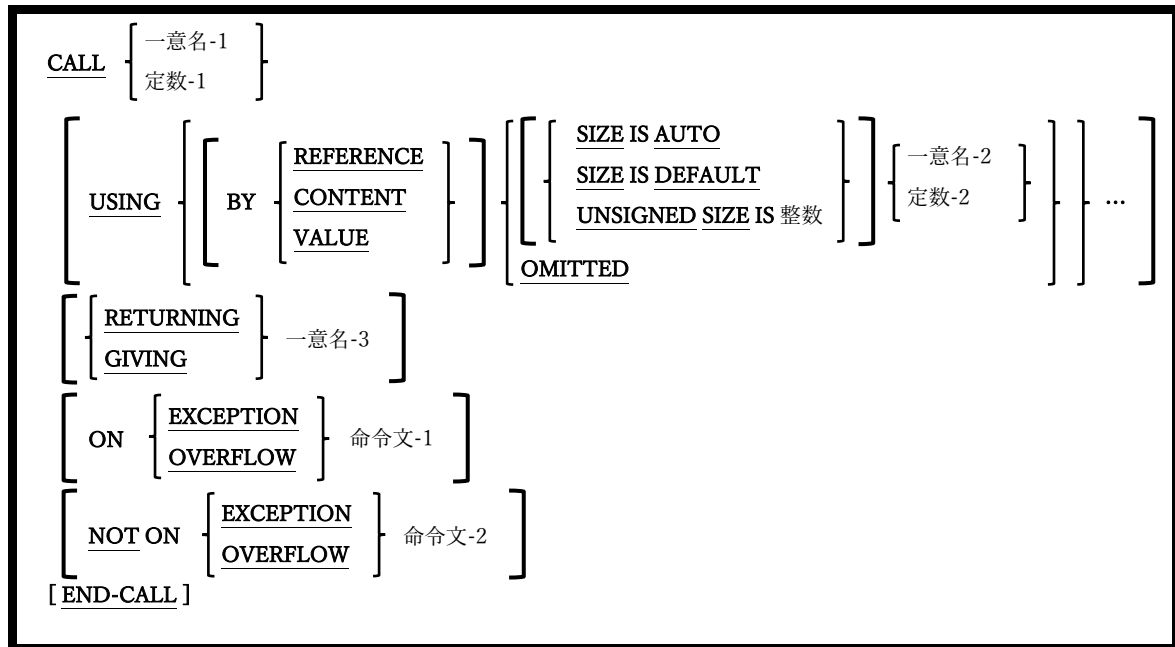
9. ストレージが割り当てられる前、またはストレージが解放された後に BASED データ項目を参照すると、予測できない結果が発生する<sup>19</sup>。

---

<sup>19</sup> COBOL 標準では、「unpredictable results - 予測不可能な結果」という用語で、予期しないまたは望ましくない動作を示し、プログラムは無効なアドレスへのアクセスを中止する可能性がある。

## 6.7. CALL

図 6-31-CALL 構文



CALL 文は、サブプログラムまたはサブルーチンと呼ばれる別のプログラムに制御を移行するために使われる。

1. サブプログラムは最終的に制御を CALL する側のプログラムに戻し、CALL 文の直後の文から実行を再開することが期待される。ただし、サブプログラムは CALL する側のプログラムに戻る必要はなく、必要に応じてプログラムの実行を自由に停止することができる。
2. EXCEPTION キーワードと OVERFLOW キーワードは同意義のものとして扱うことができる。
3. RETURNING キーワードと GIVING キーワードは同意義のものとして扱うことができる。
4. 定数-1 または identifier-1 の値は、呼び出しをするサブプログラムの記述項ポイントである。この記述項ポイントの使用の詳細については、7.1.4 および 7.1.5 で説明す

る。

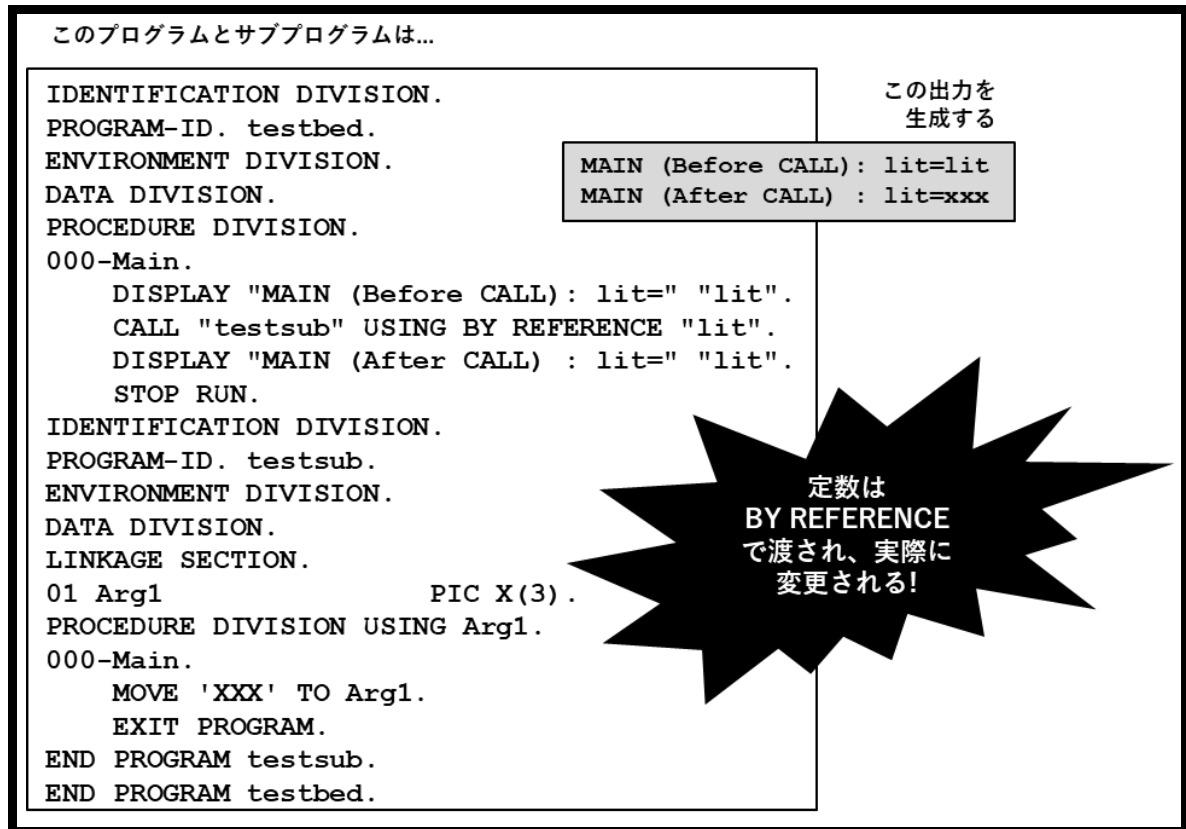
5. 一意名-1 を使ってサブルーチンを呼び出すと、ランタイムシステムに、動的にロード可能なモジュールを呼び出すよう強制される。このモジュールについては、7.1.4 で説明する。
6. ON EXCEPTION 句では、動的にロード可能なモジュールのロードが失敗した場合に実行されるコードを指定する。ON EXCEPTION を指定すると、エラーメッセージを生成してプログラムを停止する、という初期動作が上書きされ、指定したロジックへと置き換えられる。
7. NOT ON EXCEPTION 句では、動的にロード可能なモジュールのロードが成功した場合に実行されるコードを指定する。
8. USING 句では、CALL する側のプログラムからサブプログラムに渡される可能性のある引数のリストを定義する。引数が渡される方法は、BY 句によって異なる。
9. CALL されるサブプログラムが opensource COBOL プログラムであり、そのプログラムの PROGRAM-ID 句に INITIAL 属性が指定されている場合、サブプログラムが実行されるたびに、データ部の全てのデータが初期状態に復元される<sup>20</sup>。この[再]初期化動作は、INITIAL の使用(または不使用)に関係なく、サブプログラムの LOCAL-STORAGE SECTION(存在する場合)で定義されたすべてのデータに適用される。
10. BY REFERENCE 句(既定値)は引数のアドレスをサブプログラムに渡し、サブプログラムがその引数の値を変更できるようにする。引数として渡されるのが定数値であるとき、これは危険な行為となる場合がある。
11. BY CONTENT は、引数のコピーのアドレスをサブプログラムに渡す。サブプログラムが引数の値を変更した場合、CALL する側のプログラムに戻された元のバージョンは変更されない。図 6-32 に示すように、これは定数値をサブプログラムに渡すための

---

<sup>20</sup> サブプログラム内のどのエン트리ポイントが CALL されるかは関係しない。

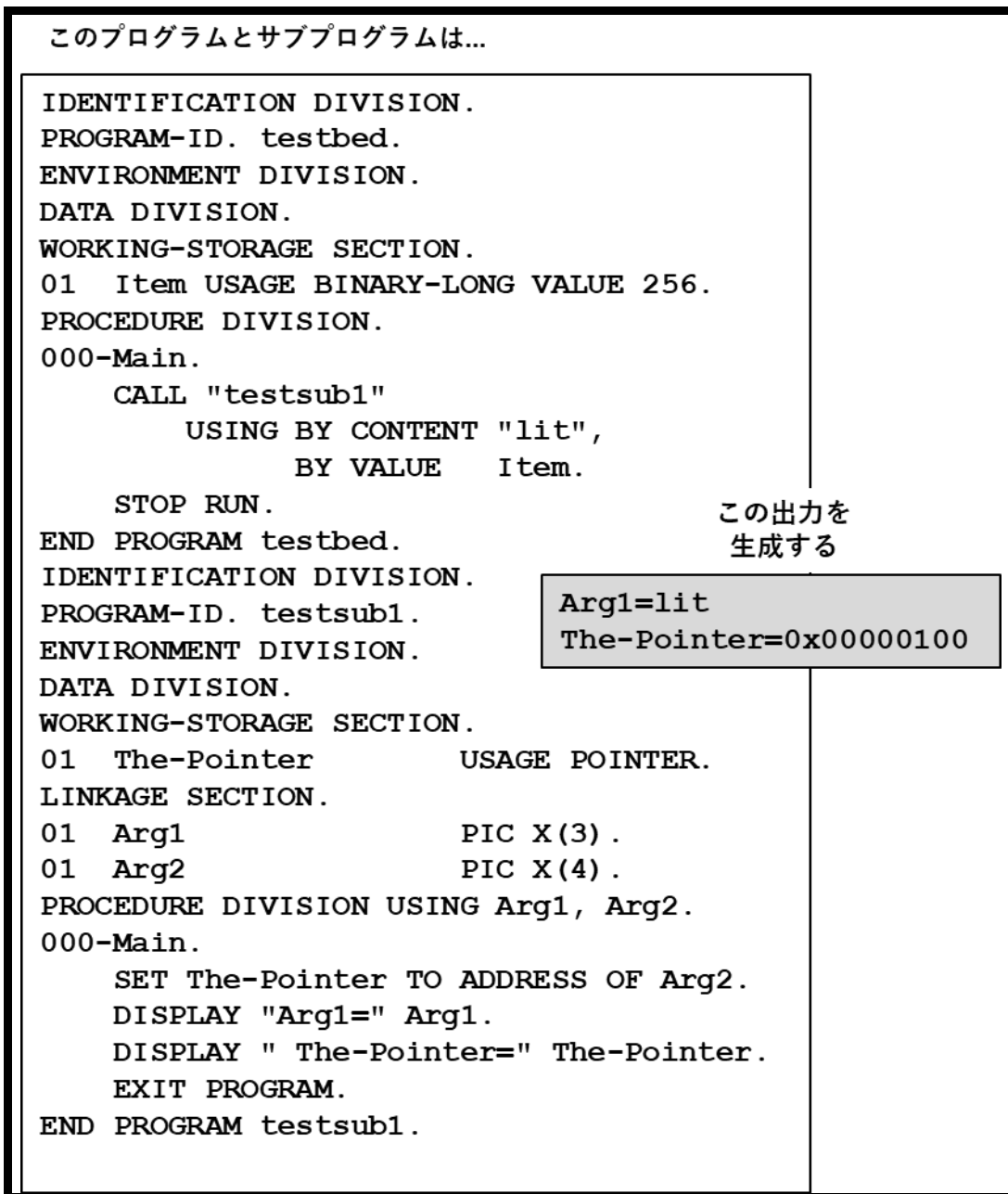
最も安全な方法である。

図 6-32-CALL BY REFERENCE 句(望ましくない影響を及ぼす場合がある)



12. BY VALUE は、引数のアドレスを引数として渡す。図 6-33 にコーディング例を示したが、サブプログラムが opensource COBOL で記述されている場合は、おそらくこのコーディングは不要である。なぜならこの機能は、C、C ++およびその他の言語との互換性を持たせるために存在するからである。

図 6-33 CALL BY VALUE 句



13. RETURNING 句では、サブルーチンが値を返すデータ項目を指定することができる。

CALL でこの句を使う場合、サブルーチンの手続き部のヘッダーに RETURNING 句を含める必要がある。もちろんサブルーチンは、BY REFERENCE によって渡された任意の引数に値を返すことができる。

14. その他詳細については 6.8(CANCEL)、6.16(ENTRY)、6.18(EXIT)、および

6.21(GOBACK)で説明する。

## 6.8. CANCEL

図 6-34-CANCEL 構文

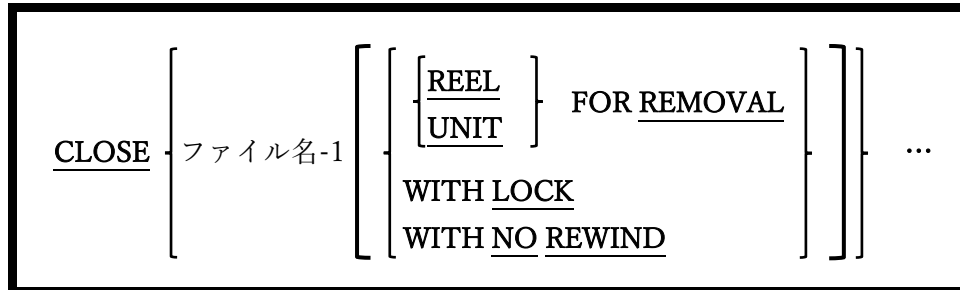
|                                                                                                 |
|-------------------------------------------------------------------------------------------------|
| <u>CANCEL</u> $\left\{ \begin{array}{l} \text{一意名-1} \\ \text{定数-1} \end{array} \right\} \dots$ |
|-------------------------------------------------------------------------------------------------|

CANCEL 文は、定数-1 または一意名-1 として指定された記述項ポイントを含む、動的にロード可能なモジュールをメモリから破棄する。

1. CANCEL によって破棄された動的にロード可能なモジュールがその後再実行されると、そのモジュールのデータ部のすべてのストレージが再び初期状態になる。

## 6.9. CLOSE

図 6-35-CLOSE 構文



CLOSE 文は、指定されたファイルまたは現在実装されているリール/ユニットへのプログラムアクセスを終了する。

1. CLOSE 文は、正常に OPEN されたファイルに対してのみ実行でき CLOSE 文は、正常に開かれたファイルに対してのみ実行できる。
2. REEL、UNIT、および NO REWIND 句は、ORGANIZATION SEQUENTIAL(LINE または RECORD BINARY)SEQUENTIAL ファイルでのみ使うことができる。REEL と UNIT という言葉は同意義で使われる場合があり、複数のリムーバブルテープ/ディスクに保存されている、または書き込まれるファイルを反映している。すべてのシステムがそのようなデバイスをサポートしているわけではないため、複数ユニットのファイルを操作できるといった opensource COBOL の特性がシステムでは機能しない場合がある。
3. REEL および UNIT 句は、SELECT 句で MULTIPLE REEL または MULTIPLE UNIT が指定されているファイルでの使用を目的としている。ランタイムシステムが複数ユニットのファイルを認識しない場合、CLOSE REEL および CLOSE UNIT 文は機能しない。
4. ファイルが閉じられると、再び正常に OPEN されるまで、ファイルに再度アクセスすることはできない。
5. OUTPUT モードまたは EXTEND モードのいずれかで OPEN されたファイルに対し



て、REEL または UNIT を使うことなく CLOSE が正常に実行されると、残りの未書込レコードバッファがファイルに書き込まれ、OPEN モードに関係なく、閉じたファイルに対して保持されていたレコードロックも解放される。閉じられたファイルは、再度 OPEN されるまで、後続の READ、WRITE、REWRITE、START、または DELETE 文で使用できなくなる。

6. CLOSE WITH LOCK は、プログラムが同じプログラム実行内でファイルを再度開いてしまうことを防いでくれる。
7. REEL または UNIT を使って CLOSE を正常に実行すると、残りの未書込レコードバッファが閉じられたファイルに書き込まれ、それらのファイルに対して保持されていたレコードロックも解放される。現在実装されているリール/ユニットは実装が解除され、次のリール/ユニットが要求される。この時ファイルは開かれたままである。

## 6.10. COMMIT

図 6-36-COMMIT 構文



COMMIT

COMMIT 文は、現在開いているすべてのファイルに対して UNLOCK を実行する。

1. 詳細については UNLOCK (6.48)の章内で説明する。

## 6.11. COMPUTE

図 6-37-COMPUTE 構文

```
COMPUTE { 一意名-1 [ROUNDED] } ... $\left\{ \begin{array}{c} \underline{\text{EQUAL}} \\ = \end{array} \right\}$ 算術式
[ON SIZE ERROR 命令文-1]
[NOT ON SIZE ERROR 命令文-2]
[END-COMPUTE]
```

COMPUTE 文は、ADD、SUBTRACT、MULTIPLY、および DIVIDE 文といった、厄介で混乱を招く恐れのある構文を使用する代わりに、たった一文で複雑な算術演算を簡単に実行することができる。

1. 単語の EQUAL と等号(=)は同意義のものとして扱うことができる。
2. ON SIZE ERROR、NOT ON SIZE ERROR、および ROUNDED 句はコード化されており、ADD 文で使われている同名義の句と同様に動作する(6.5.1 を参照)。

## 6.12. CONTINUE

図 6-38-CONTINUE 構文

CONTINUE

CONTINUE 文は動作がないためアクションを実行しない。

1. CONTINUE 文は、IF 文(6.23)とともに、まだ必要とされていないか、または未設計の条件付きで実行されるコードのプレースホルダーとして多用される。次の二つの文は同等である。CONTINUE 文を使うことで、今後コード挿入の必要があるかもしれない場所をマークする。

| 必要最低限のコーディング                                                                                                                                                                      | CONTINUE 文を使ったコーディング<br>(今後コードが必要になる可能性のある箇所も示す)                                                                                                                                                                                                                      |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>IF A = 1   IF B = 1     DISPLAY „A=1 &amp; B=1“ END-DISPLAY   END-IF ELSE   IF A = 2     IF B = 2       DISPLAY „A=2 &amp; B=2“ END-DISPLAY     END-IF   END-IF END-IF</pre> | <pre>IF A = 1   IF B = 1     DISPLAY „A=1 &amp; B=1“ END-DISPLAY     ELSE       CONTINUE     END-IF ELSE   IF A = 2     IF B = 2       DISPLAY „A=2 &amp; B=2“ END-DISPLAY       ELSE         CONTINUE       END-IF     ELSE       CONTINUE     END-IF   END-IF</pre> |

上記のようなコーディングは、一般的に個人の嗜好やウェブサイトのコーディング基準の問題である。オブジェクトコード自体に違いはないため、実行時の動作効率には関係しない(「コーディングが効率的であるか」の一点だけ)。

2. CONTINUE のもう一つの IF 文の使用法は、IF 文でコーディングされた条件式での NOT の使用を回避することで、これも個人的および/またはウェブサイト標準におけ

る問題である。例を以下に示す。

| CONTINUE 文 なし                                                                                         | CONTINUE 文 あり                                                                                                    |
|-------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| <pre> IF Action-Flag NOT = 'I' AND 'U'   DISPLAY 'Invalid Action-Flag'   EXIT PARAGRAPH END-IF </pre> | <pre> IF Action-Flag = 'I' OR 'U'   CONTINUE ELSE   DISPLAY 'Invalid Action-Flag'   EXIT PARAGRAPH END-IF </pre> |

COBOL(opensource COBOL を含む)では条件式が省略形で処理されるため、左側の例の条件式は短縮版となっている。

**IF Action-Flag NOT = 'I' AND Action-Flag NOT = 'U'**

プログラマの多くは、「IF」を(誤って)「**IF Action-Flag NOT = 'I' OR 'U'**」としてコーディングしていた。これにより、実行時に問題が発生することは避けられない。

従ってプログラマは、少し長くても右側の例のコードの方が読みやすいと考えている。

## 6.13. DELETE

図 6-39-DELETE 構文

```
DELETE ファイル名 RECORD
 [INVALID KEY 命令文-1]
 [NOT INVALID KEY 命令文-2]
 [END-DELETE]
```

DELETE 文は、ORGANIZATION RELATIVE または ORGANIZATION INDEXED ファイルから論理的にレコードを削除する。

1. ACCESS MODE IS SEQUENTIAL であるファイルには、INVALID KEY 句と NOT INVALID KEY 句を指定できない。
2. INVALID KEY 句には、DELETE の失敗に対応できる機能があり、NOT INVALID KEY 句は、DELETE の成功時に実行するアクションをプログラムが指定する機能を持つ。
3. ORGANIZATION のファイル名は、RELATIVE または INDEXED でなければならない。
4. SEQUENTIAL アクセスモードの RELATIVE または INDEXED ファイルは、DELETE 文の実行前にファイル名に対して実行された最後の入出力文が、正常に実行された READ 文である必要があり、削除されるレコードを識別している。
5. RELATIVE ファイルの ACCESS MODE が RANDOM または DYNAMIC の場合、削除されるレコードは、相対レコード番号が RELATIVEKEY として指定された現在の項目値である。
6. INDEXED ファイルの ACCESS MODE が RANDOM または DYNAMIC の場合、削除されるレコードは、主キーが RECORD KEY として指定された現在の項目値である。

7. RELATIVE KEY または RECORD KEY の値によって削除するように指定されたレコードが、アクセスモードの RANDOM ファイルまたは DYNAMIC ファイルに存在しない場合、INVALID KEY 条件によって INVALID KEY 句を介して処理できる。これは4項に記述したように、ACCESS MODE SEQUENTIAL ファイルには存在しない条件である。ACCESS MODE SEQUENTIAL ファイルでの DELETE 文の失敗は、DECLARATIVES を介してのみ「処理」することが可能である。

## 6.14. DISPLAY

### 6.14.1. DISPLAY 文の書き方 1 — UPON CONSOLE

図 6-40-DISPLAY 構文(UPON CONSOLE)

|                                                                                                                                                                                                                      |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre><u>DISPLAY</u>  { 一意名-1<br/>            定数-1  }  ...<br/>            [ <u>UPON</u> ニーモニック名 ]<br/>            [ <u>WITH NO ADVANCING</u> ]<br/>            [ 例外処理 ]<br/>            [ <u>END-DISPLAY</u> ]</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

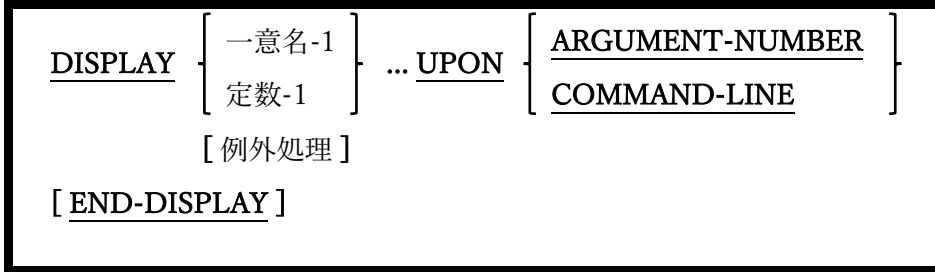
プログラムが開始されたシェルまたはコンソールウィンドウに、指定された一意名の内容や定数値を表示する。テキストは、次に使用可能な行の 1 列目から表示される。すべての画面行に既にテキストが表示されていた場合、画面は 1 行上にスクロールし、テキストは最後の行に表示される。

1. UPON 句が指定されていない場合、UPON CONSOLE が指定されたとみなす。
2. 指定するニーモニック名は、CONSOLE、CRT、PRINTER、またはこれらのうち 1 つに関連する特殊名段落内のユーザ定義のニーモニック名である必要がある(4.1.4 を参照)。このようなニーモニックはすべて、プログラムの実行元であるシェル(UNIX)またはコンソールウィンドウ(Windows)といった同じ宛先を指定します。
3. NO ADVANCING 句を使うと、コンソールディスプレイの最後に追加される通常の行頭復帰/改行順序が抑制される。



### 6.14.2. DISPLAY 文の書き方 2 — コマンドライン引数へのアクセス

図 6-41-DISPLAY 構文(コマンドライン引数へのアクセス)

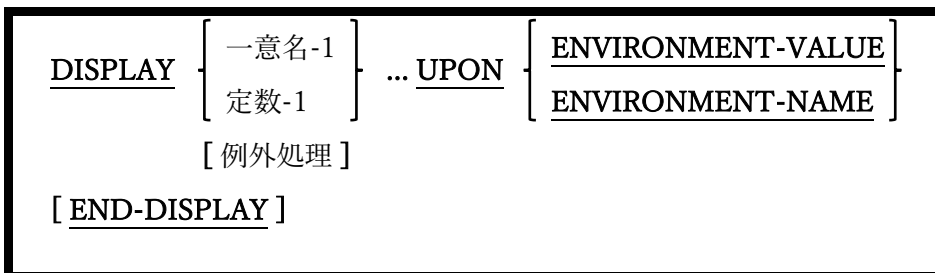


後続の ACCEPT によって取得されるコマンドライン引数番号を指定したり、コマンドライン引数自体に新しい値を指定することができる。

1. DISPLAY...UPON COMMAND-LINE を実行すると、後続の ACCEPT...FROM COMMAND-LINE 文に影響する(その後に DISPLAY された値が返される)が、後続の ACCEPT...FROM ARGUMENT-VALUE 文には影響せず、元のプログラム実行パラメータを返す。

### 6.14.3. DISPLAY 文の書き方 3 — 環境変数へのアクセスまたは設定

図 6-42-DISPLAY 構文(環境変数へのアクセス/設定)



環境変数を作成または変更するために使われる。

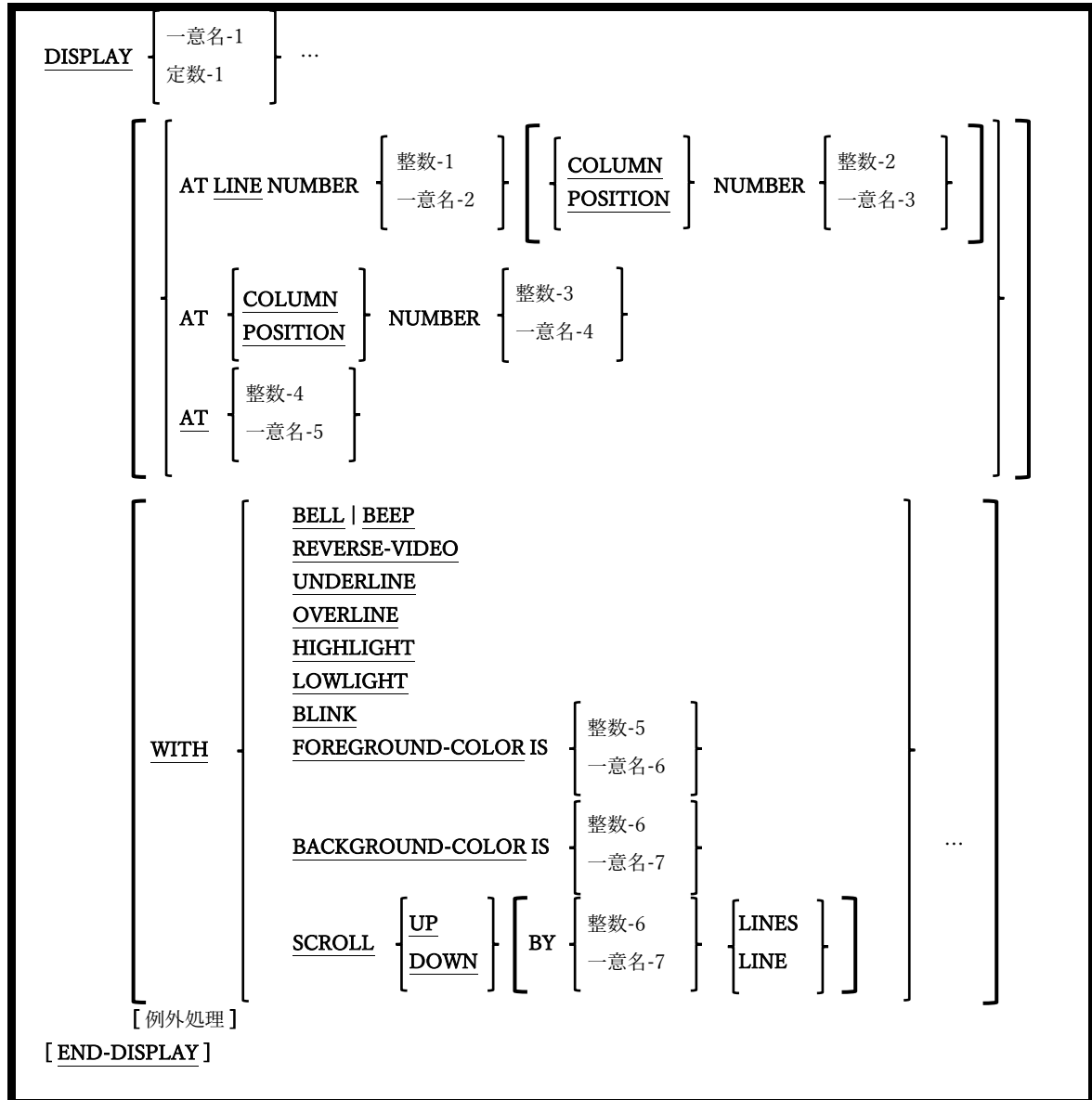
1. 環境変数を作成または変更するには、二つの DISPLAY 文が必須となり、次の手順で実行する必要がある。

**DISPLAY**`environment-variable-name UPON ENVIRONMENT-NAME`**END-DISPLAY****DISPLAY**`environment-variable-value UPON ENVIRONMENT-VALUE`**END-DISPLAY**

2. opensource COBOL プログラム内から作成または変更された環境変数は、そのプログラムによって生成されたサブシェルプロセス(つまり、CALL“SYSTEM”)では使用できるが、opensource COBOL プログラムを開始したシェルまたはコンソールウィンドウからは認識されない。
3. DISPLAY の代わりに SET ENVIRONMENT(6.39.1)を使用して環境変数を設定する方がはるかに簡単である。

## 6.14.4. DISPLAY 文の書き方 4 — 画面データ

図 6-43-DISPLAY 構文(画面データ)



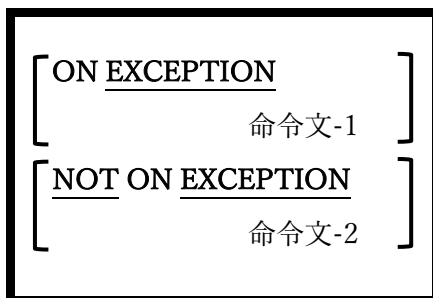
形式化された画面にデータを表示する。

- 一意名-1 が画面節で定義されている場合、すべてのカーソル位置(AT)および属性指定(WITH)も画面節の定義から取得され、DISPLAY 文で指定されたものはすべて無視される。画面節で定義されていないデータ項目を表示する場合のみ、ATおよびWITHオプションを使用する。

2. AT 句では、データが画面に表示される前に、カーソルを画面上の特定の場所に配置することができる。定数-3/一意名-4の値は4桁である必要があり、最初の2桁はカーソルを配置する行、最後の2桁は列を示す。
3. SCROLL オプションについては、6.4.4(ACCEPT 文の書き方 4 — 画面データの取得)で説明している。
4. WITH オプションについては、5.6(画面記述)で説明している。

#### 6.14.5. DISPLAY 文の例外処理

図 6-44-例外処理構文(DISPLAY)



DISPLAY 文のすべての書き方で使用可能な EXCEPTION 句と NOT EXCEPTION 句を使うことで、DISPLAY 文の失敗、成功時のそれぞれに実行されるコードを指定することができる。DISPLAY 文ではリターンコードやステータスフラグを設定しないため、これが成功と失敗を検出する唯一の方法となっている。

## 6.15. DIVIDE

### 6.15.1. DIVIDE 文の書き方 1 — DIVIDE INTO

図 6-45-DIVIDE INTO 構文

```
DIVIDE { 一意名-1
 定数-1 } INTO { 一意名-2 [ROUNDED] }...
 [ON SIZE ERROR 命令文-1]
 [NOT ON SIZE ERROR 命令文-2]
 [END-DIVIDE]
```

指定された値を一つ以上のデータ項目に分割し、それらの各データ項目を一意名-1 または定数-1 値で割った結果に置き換える。除算の余りは破棄される。

1. 一意名-1 および一意名-2 は、編集不可の数値データ項目でなければならない。
2. 定数-1 は数字定数でなければならない。
3. ON SIZE ERROR、NOT ON SIZE ERROR、および ROUNDED 句はコード化されており、ADD 文で使われている同名義の句と同様に動作する(6.5 を参照)。
4. 一意名-1 /定数-1 の値がゼロの時、SIZE ERROR 条件が発生する。除算の結果、小数点の左側に、受け取り項目で使用可能な数を超える桁数が必要な場合も同様である。

### 6.15.2. DIVIDE 文の書き方 2 — DIVIDE INTO GIVING

図 6-46-DIVIDE INTO GIVING 構文

|                                                                                                                                                                                                                                                                                                                                                              |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><u>DIVIDE</u> <math>\left[ \begin{array}{c} \text{一意名-1} \\ \text{定数-1} \end{array} \right]</math> <u>INTO</u> <math>\left[ \begin{array}{c} \text{一意名-2} \\ \text{定数-2} \end{array} \right]</math> <u>GIVING</u> { 一意名-3 [ <u>ROUNDED</u> ] }...<br/>[ <u>ON SIZE ERROR</u> 命令文-1 ]<br/>[ <u>NOT ON SIZE ERROR</u> 命令文-2 ]<br/>[ <u>END-DIVIDE</u> ]</p> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

指定された値(一意名-1 / 定数-1)を別の値(一意名-2 / 定数-2)に分割し、一つ以上の受け取りデータ項目(一意名-3...)の内容を除算結果に置き換える。除算の余りは破棄される。

1. 一意名-1 および一意名-2 は、編集不可の数値データ項目でなければならない。
2. 一意名-3 は数値データ項目でなければならないが、編集可能な場合もある。
3. 定数-1 と定数-2 は数字定数でなければならない。
4. ON SIZE ERROR、NOT ON SIZE ERROR、および ROUNDED 句はコード化されており、ADD 文で使われている同名義の句と同様に動作する(6.5 を参照)。
5. 一意名-1 / 定数-1 の値がゼロの時、SIZE ERROR 条件が発生する。除算の結果、小数点の左側に、受け取り項目での使用可能な数を超える桁数が必要な場合も同様である。

### 6.15.3. DIVIDE 文の書き方 3 — DIVIDE BY GIVING

図 6-47-DIVIDE BY GIVING 構文

DIVIDE  $\left[ \begin{array}{c} \text{一意名-1} \\ \text{定数-1} \end{array} \right] \text{ BY } \left[ \begin{array}{c} \text{一意名-2} \\ \text{定数-2} \end{array} \right] \text{ GIVING } \{ \text{一意名-3 [ ROUNDED ] } \} \dots$   
[ ON SIZE ERROR 命令文-1 ]  
[ NOT ON SIZE ERROR 命令文-2 ]  
[ END-DIVIDE ]

指定された値(一意名-1 / 定数-1)を別の値(一意名-2 / 定数-2)で除算し、一つ以上の受け取りデータ項目(一意名-3...)の内容を除算結果に置き換える。除算の余りは破棄される。

1. 一意名-1 および一意名-2 は、編集不可の数値データ項目でなければならない。
2. 一意名-3 は数値データ項目でなければならないが、編集可能な場合もある。
3. 定数-1 と定数-2 は数字定数でなければならない。
4. ON SIZE ERROR、NOT ON SIZE ERROR、および ROUNDED 句はコード化されており、ADD 文で使われている同名義の句と同様に動作する(6.5 を参照)。
5. 一意名-1 / 定数-1 の値がゼロの時、SIZE ERROR 条件が発生する。除算の結果、小数点の左側に、受け取り項目での使用可能な数を超える桁数が必要な場合も同様である。

#### 6.15.4. DIVIDE 文の書き方 4 — DIVIDE INTO REMAINDER

図 6-48-DIVIDE INTO REMAINDER 構文

```
DIVIDE { 一意名-1
 定数-1 } INTO { 一意名-2
 定数-2 } GIVING { 一意名-3 [ROUNDED] }...
 REMAINDER 一意名-4
 [ON SIZE ERROR 命令文-1]
 [NOT ON SIZE ERROR 命令文-2]
 [END-DIVIDE]
```

指定された値(一意名-1 / 定数-1)を別の値(一意名-2 / 定数-2)に分割し、一つの受け取りデータ項目(一意名-3...)の内容を除算結果に置き換える。除算の余りは一意名-4 に格納される。

1. 一意名-1 および一意名-2 は、編集不可の数値データ項目でなければならない。
2. 一意名-3 と一意名-4 は数値データ項目でなければならないが、編集可能な場合もある。
3. 定数-1 と定数-2 は数字定数でなければならない。
4. ON SIZE ERROR、NOT ON SIZE ERROR、および ROUNDED 句はコード化されており、ADD 文で使われている同名義の句と同様に動作する(6.5 を参照)。
5. 一意名-1 / 定数-1 の値がゼロの時、SIZE ERROR 条件が発生する。除算の結果、小数点の左側に、受け取り項目での使用可能な数を超える桁数が必要な場合も同様である。



### 6.15.5. DIVIDE 文の書き方 5 — DIVIDE BY REMAINDER

図 6-49-DIVIDE BY REMAINDER 構文

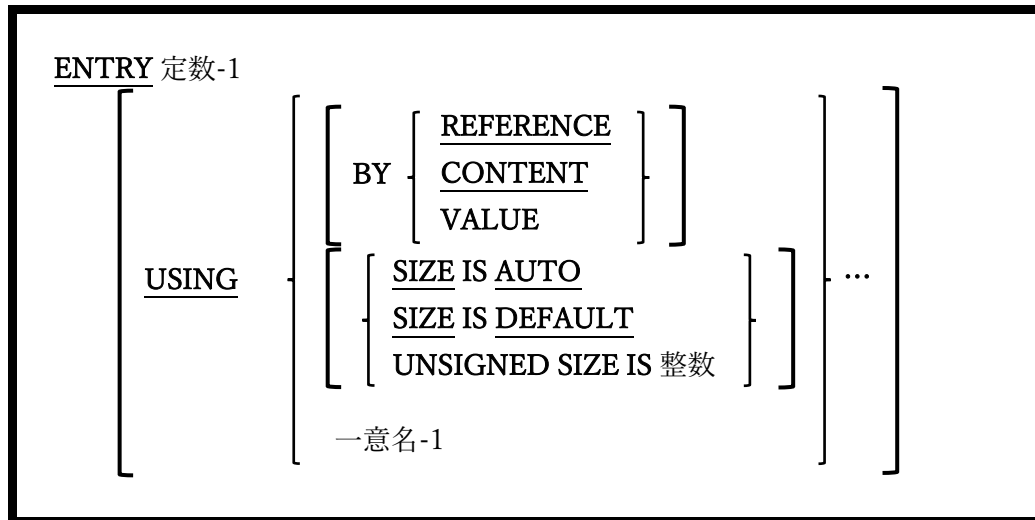
```
DIVIDE {一意名-1 / 定数-1} BY {一意名-2 / 定数-2} GIVING {一意名-3 [ROUNDED] }...
 REMAINDER 一意名-4
 [ON SIZE ERROR 命令文-1]
 [NOT ON SIZE ERROR 命令文-2]
 [END-DIVIDE]
```

指定された値(一意名-1 / 定数-1)を別の値(一意名-2 / 定数-2)で除算し、一つの受け取りデータ項目(一意名-3...)の内容を除算結果に置き換える。除算の余りは一意名-4 に格納される。

1. 一意名-1 および一意名-2 は、編集不可の数値データ項目でなければならない。
2. 一意名-3 と一意名-4 は数値データ項目でなければならないが、編集可能な場合もある。
3. 定数-1 と定数-2 は数字定数でなければならない。
4. ON SIZE ERROR、NOT ON SIZE ERROR、および ROUNDED 句はコード化されており、ADD 文で使われている同名義の句と同様に動作する(6.5 を参照)。
5. 一意名-1 / 定数-1 の値がゼロの時、SIZE ERROR 条件が発生する。除算の結果、小数点の左側に、受け取り項目での使用可能な数を超える桁数が必要な場合も同様である。

## 6.16. ENTRY

図 6-50-ENTRY 構文

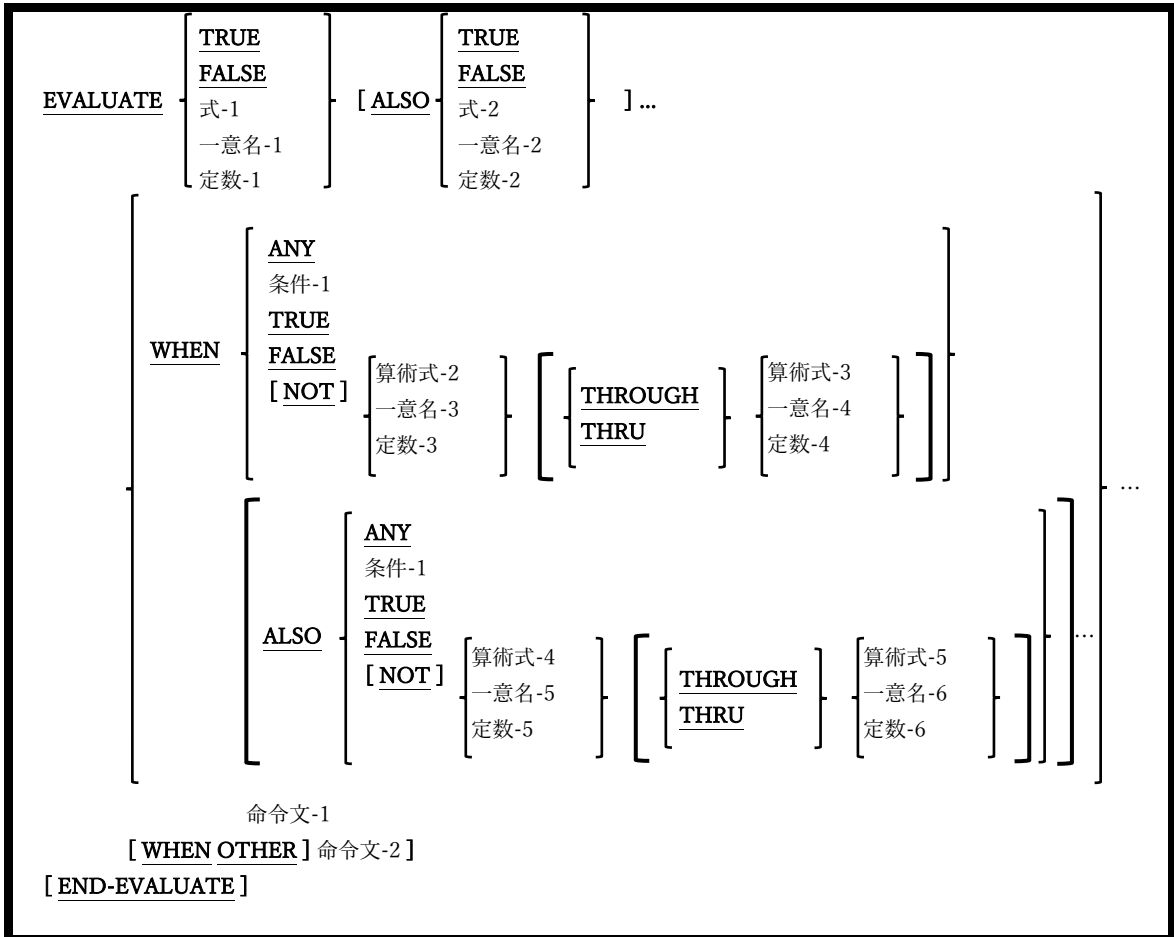


ENTRY 文は、サブルーチンが予期する引数とともに、サブルーチンへの代替記述項ポイントを定義するために使用する。

1. ネストされたサブプログラムで ENTRY 文を使うことはできない(2.1 を参照)。
2. ENTRY 文の USING 句は、サブルーチンを呼び出す CALL 文の USING 句と一致する。
3. 定数-1 の値によって、サブルーチンの記述項ポイント名を指定する。ENTRY 文で指定されているように、(大文字と小文字の使用に関して)CALL 文で正確に指定する必要がある。

# 6.17. EVALUATE

図 6-51-EVALUATE 構文



EVALUATE 文では、さまざまな状況に合わせて実行する必要がある処理を定義する。

1. 予約語の THRU と THROUGH は同意義のものとして扱うことができる。
2. THROUGH を使う場合、THROUGH 句に関連する値(算術式-n、一意名-n、および/または定数-n)は同じクラスである必要がある。 例：

**Legal:**  
(3 + Years-Of-Service) THROUGH 99  
"A" THRU "Z"  
X'00' THRU X'1F'  
15.7 THROUGH 19.4

**Not Legal:**  
0 THRU "A"  
Last-Name THRU Zip-Code (Assuming Last-Name is  
PIC X and Zip-Code is PIC 9)

3. EVALUATE 文の後、最初の WHEN 句の前に指定された値は選択サブジェクトと呼ばれ、各 WHEN 句の後に指定された値は選択オブジェクトと呼ばれる。
4. 各 WHEN 句には、EVALUATE 文の選択サブジェクトと同じ数の選択オブジェクトが必要である。
5. 各 EVALUATE 句の選択サブジェクトは、選択オブジェクトに対応する各 WHEN 句と等しいかどうかテストされる。
6. 5 項のテストで等しいと判断され、結果が TRUE である最初の WHEN 句では、命令文が実行される。
7. 5 項のテストで WHEN 句との同等性はなく、結果が TRUE である場合、WHEN OTHER 句に関連する命令文(命令文-2)が実行される。WHEN OTHER 句がない場合、制御は EVALUATE 文に続く次の文へ移る。
8. WHEN または WHEN OTHER 句の命令文が実行されると、制御は EVALUATE 文に続く次の文へ移る。
9. ANY の選択オブジェクトを使うと、ANY と一致する選択サブジェクトと自動的に合致する。

ここで、EVALUATE 文の利便性がわかる事例を示す。一日の平均残高[ADB]に基づいて口座に支払われる利息を計算するプログラムが開発され、プログラムは以下のように定義されている。

1. 平均残高が 1000 ドル未満の場合、有利子当座預金口座には利息がつかない。平均残高が 1,000 ドルから 1,499.99 ドルの有利子当座預金口座はその 1%、1500 ドル以上はその 1.5%を利子として受け取る。
2. 定期預金口座は、平均残高が 10,000 ドルまでは 1.5%、10,000 ドル以上は 1.75%の利息が適用される。

3. プラチナ普通預金口座は、平均残高に関係なく 2%の利子を受け取る。
4. 上記以外の種類の口座には利子が適用されない。

これらのルールを適用した「EVALUATE」実装をテストするために opensource COBOL プログラムのサンプルを次に示す。挿入図はプログラムからの出力結果である。

図 6-52-EVALUATE 文のデモプログラム

```

>>SOURCE FORMAT FREE
IDENTIFICATION DIVISION,
PROGRAM-ID. evaldemo.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 Account-Type PIC X(1).
 88 Interest-Bearing-Checking VALUE 'C'.
 88 Statement-Savings VALUE 's'.
 88 Platinum-Savings VALUE 'p'.
01 ADB-Char PIC X(10).
01 Ave-Daily-Balance PIC 9(7)V99.
01 Formatted-Amount PIC Z(6)9.99.
01 Interest-Amount PIC 9(7)V99.
PROCEDURE DIVISION,
000-Main.
 PERFORM FOREVER
 DISPLAY "Enter Account Type (c,s,p,other): " WITH NO ADVANCING
 ACCEPT Account-Type
 IF Account-Type = SPACES
 STOP RUN
 END-IF
 DISPLAY "Enter Ave Daily Balance (nnnnnnn.nn): " WITH NO ADVANCING
 ACCEPT ADB-char
 MOVE FUNCTION NUMVAL(ADB-Char) TO Ave-Daily-Balance
 EVALUATE TRUE ALSO Ave-Daily-balance
 WHEN Interest-Bearing-Checking ALSO 0.00 THRU 999.99
 MOVE 0 TO Interest-Amount
 WHEN Interest-Bearing-Checking ALSO 1000.00 THRU 1499.99
 COMPUTE Interest-Amount ROUNDED = 0.01 * Ave-Daily-Balance
 WHEN Interest-Bearing-Checking ALSO ANY
 COMPUTE Interest-Amount ROUNDED = 0.015 * Ave-Daily-Balance
 WHEN Statement-Savings ALSO 0.00 THRU 10000.00
 COMPUTE Interest-Amount ROUNDED = 0.015 * Ave-Daily-Balance
 WHEN Statement-Savings ALSO ANY
 COMPUTE Interest-Amount ROUNDED = 0.015 * Ave-Daily-Balance
 + 0.175 * (Ave-daily-Balance - 10000)
 WHEN Platinum-Savings ALSO ANY
 COMPUTE Interest-Amount ROUNDED = 0.020 * Ave-Daily-Balance
 WHEN OTHER
 MOVE 0 TO Interest-Amount
 END-EVALUATE
 MOVE Interest-Amount TO Formatted-Amount
 DISPLAY "Accrued Interest = " Formatted-Amount
 END-PERFORM

```

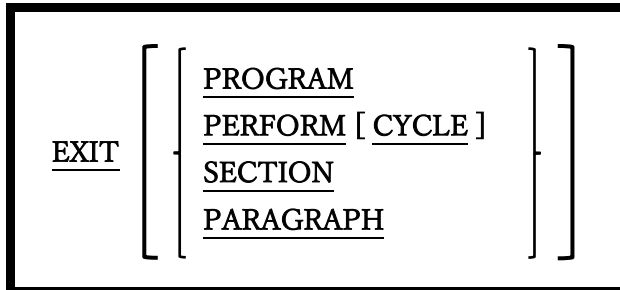
```

Enter Account Type(c,s,p,other): c
Enter Ave Daily Balance(nnnnnnn.nn): 250
Accrued Interest = 0.00
Enter Account Type(c,s,p,other): c
Enter Ave daily Balance(nnnnnnn.nn): 1250
Accrued Interest = 12.50
Enter Account Type(c,s,p,other): c
Enter Ave Daily Balance(nnnnnnn.nn): 1899.99
Accrued Interest = 28.50
Enter Account Type(c,s,p,other): s
Enter Ave Daily Balance(nnnnnnn.nn): 22000.00
Accrued Interest = 2430.00
Enter Account Type(c,s,p,other): p
Enter Ave Daily Balance(nnnnnnn.nn): 1.98
Accrued Interest = 0.04

```

## 6.18. EXIT

図 6-53-EXIT 構文



EXIT 文は多様な目的に使用できる文である。一連のプロシージャに共通のエンドポイントを提供したり、インライン PERFORM、段落、または節を終了したり、呼び出されたプログラムの論理的な終了を示す。

1. 「EXIT」文をオプションの句を指定せずに使用すると、一連のプロシージャに共通の「GO TO」エンドポイントを提供する。

図 6-54-EXIT 文

```

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.
 IF Input-Rec of Input-File = SPACES
 EXIT PARAGRAPH. *> IGNORE BLANK RECORDS!
 process the record just read

```

2. EXIT 文を使う場合、それを扱う段落内で唯一の文である必要がある。

3. EXIT 文は操作不要である(CONTINUE 文とよく似ている)。
4. EXIT PARAGRAPH 文は、現在の段落の終わりを過ぎた時点に制御を移すが、EXIT SECTION 文は、現在の節の最後の段落を過ぎた時点に制御を移す。

EXIT PARAGRAPH または EXIT SECTION が手続き型 PERFORM(6.32.1)の範囲内の段落にある場合、制御は PERFORM に戻され、TIMES、VARYING、および/または UNTIL 句での評価が行われる。EXIT PARAGRAPH または EXIT SECTION が手続き型 PERFORM の範囲外にある場合、制御は次の段落(EXIT PARAGRAPH)または節(EXIT SECTION)の最初の実行可能な文に移る。図 6-55 は、EXIT PARAGRAPH 文を使って、GO TO なしで図 6-54 の例をコーディングする方法を示している。

図 6-55-EXIT PARAGRAPH 文

```
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.
 IF Input-Rec of Input-File = SPACES
 EXIT PARAGRAPH. *> IGNORE BLANK RECORDS!
 process the record just read
```

5. EXIT PERFORM および EXIT PERFORM CYCLE 文は、インライン PERFORM 文 (6.32.2)と組み合わせて使うことを目的としている。



6. EXIT PERFORM CYCLE は、インライン PERFORM の現在の繰り返しを終了し、別のサイクルを実行する必要があるかどうかを判断するために、TIMES、VARYING、および/または UNTIL 句を制御する。
7. EXIT PERFORM は、インライン PERFORM を完全に終了し、PERFORM に続く最初の文に制御を移す。図 6-56 は、図 6-54 の例に対する最終変更を示していて、インライン PERFORM 文と EXIT PERFORM 文を使うことによって処理を確実に簡素化できる。

図 6-56-EXIT PERFORM 文

```
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
```

8. 最後に、EXIT PROGRAM 文は、サブルーチン(つまり、別のプログラムによって CALL されているプログラム)の実行を終了し、CALL に続く文の CALL する側のプログラムに戻る。メインプログラムによって実行された場合は、EXIT PROGRAM 文は機能しない。COBOL2002 標準は、COBOL 言語に共通の拡張を行った。それが GOBACK 文(6.21)であり、EXIT PROGRAM の代わりとして検討すべきである。

## 6.19. FREE

図 6-57-FREE 構文

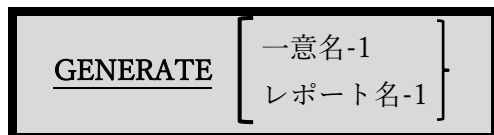
```
FREE { [ADDRESS OF] 一意名-1 } …
```

FREE 文は、ALLOCATE 文(6.6)によってプログラムに割り当てられていたメモリを解放する。

1. 一意名-1 は、USAGE POINTER データ項目または BASED 属性を持つ 01 レベルのデータ項目である必要がある。
2. 一意名-1 が USAGE POINTER データ項目であり、有効なアドレスが含まれている場合、FREE 文はポインタが参照するメモリブロックを解放する。更に、アドレスを提供するためにポインタが使用された BASED データ項目は、基準でなくなり使えなくなる。一意名-1 に有効なアドレスが含まれていなかった場合、アクションは実行されない。
3. 一意名-1 が BASED データ項目であり、そのデータ項目が現在の基準となっている場合(つまり、現在メモリが割り当てられている場合)、メモリが解放され、一意名-1 は基準でなくなり、使えなくなる。一意名-1 が基準になっていない場合、アクションは実行されない。
4. ADDRESS OF 句は、FREE 文に特別な関数を追加しない。

## 6.20. GENERATE

図 6-58-GENERATE 構文



GENERATE 文は、opensource COBOL コンパイラによって構文的には認識されるが、RWCS(COBOL Report Writer)は現在 opensource COBOL でサポートされていないため、機能しない。

## 6.21. GOBACK

図 6-59-GOBACK 構文



GOBACK

GOBACK 文は、実行中のプログラムを論理的に終了するために使用する。

1. サブルーチン(つまり、CALL されたプログラム)内で実行された場合、GOBACK は制御を CALL に続く文の CALL する側のプログラムに戻す。
2. メインプログラム内で実行された場合、GOBACK は STOP RUN 文として機能する(6.42)。

## 6.22. GO TO

### 6.22.1. GO TO 文の書き方 1 — GO TO

図 6-60-GOTO 構文

GO TO 手続き名

プログラム内の制御を指定されたプロシージャ名へ無条件に移す。

1. 指定されたプロシージャ名が SECTION の場合、制御はその節の最初の段落に移る。

### 6.22.2. GO TO 文の書き方 2 — GO TO DEPENDING ON

図 6-61-GOTO DEPENDING ON 構文

GO TO 手続き名-1 ...  
DEPENDING ON 一意名-1

文で指定された一意名の数値に応じて、指定された手続き名のいずれかに制御を移す。

1. 指定された一意名-1 の PICTURE および/または USAGE 句は、数値であり、編集できない、できれば符号なし整数データ項目として定義するようなものでなければならない。
2. 一意名-1 の値が 1 の場合、制御は最初に指定された手続き名に移され、値が 2 の場合、制御は 2 番目の手続き名やその他に移る。
3. 一意名-1 の値が 1 未満であるか、GO TO 文で指定された手続き名の総数を超過している場合、制御は GO TO に続く次の文に移る。
4. 次の表は、実際の適用状況下で GO TO DEPENDING ON をどのように使うかを示し、IF と EVALUATE の二つと比較している。

図 6-62-GOTO DEPENDING ON vs IF vs EVALUATE

| GO TO DEPENDING ON                                                                                                                                                                                                                                                                                                                                                                                                                                           | IF                                                                                                                                                                                                          | EVALUATE                                                                                                                                                                                                     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>GO TO PROCESS-ACCT-TYPE-1</b><br><b>PROCESS-ACCT-TYPE-2</b><br><b>PROCESS-ACCT-TYPE-3</b><br><b>DEPENDING ON ACCT-TYPE.</b><br>無効アカウントタイプの処理コード<br><b>GO TO DONE-WITH-ACCT-TYPE.</b><br><b>PROCESS-ACCT-TYPE-1.</b><br>アカウントタイプ 1 の処理コード<br><b>GO TO DONE-WITH-ACCT-TYPE.</b><br><b>PROCESS-ACCT-TYPE-2.</b><br>アカウントタイプ 2 の処理コード<br><b>GO TO DONE-WITH-ACCT-TYPE.</b><br><b>PROCESS-ACCT-TYPE-3.</b><br>アカウントタイプ 3 の処理コード<br><b>DONE-WITH-ACCT-TYPE.</b> | <b>IF ACCT-TYPE = 1</b><br>アカウントタイプ 1 の処理コード<br><b>ELSE IF ACCT-TYPE = 2</b><br>アカウントタイプ 2 の処理コード<br><b>ELSE IF ACCT-TYPE = 3</b><br>アカウントタイプ 3 の処理コード<br><b>ELSE</b><br>無効アカウントタイプの処理コード<br><b>END-IF.</b> | <b>EVALUATE ACCT-TYPE</b><br><b>WHEN 1</b><br>アカウントタイプ 1 の処理コード<br><b>WHEN 2</b><br>アカウントタイプ 2 の処理コード<br><b>WHEN 3</b><br>アカウントタイプ 3 の処理コード<br><b>WHEN OTHER</b><br>無効アカウントタイプの処理コード<br><b>END-EVALUATE.</b> |

「現代のプログラミング哲学」で EVALUATE 文が好まれるのは間違いない。興味深いことに、IF 文と EVALUATE 文によって生成されたコードは実質的に同じである。新しいものは、必ずしも違いを意味するわけではなく、より良いと見なされる場合もある。

## 6.23. IF

図 6-63-IF 構文

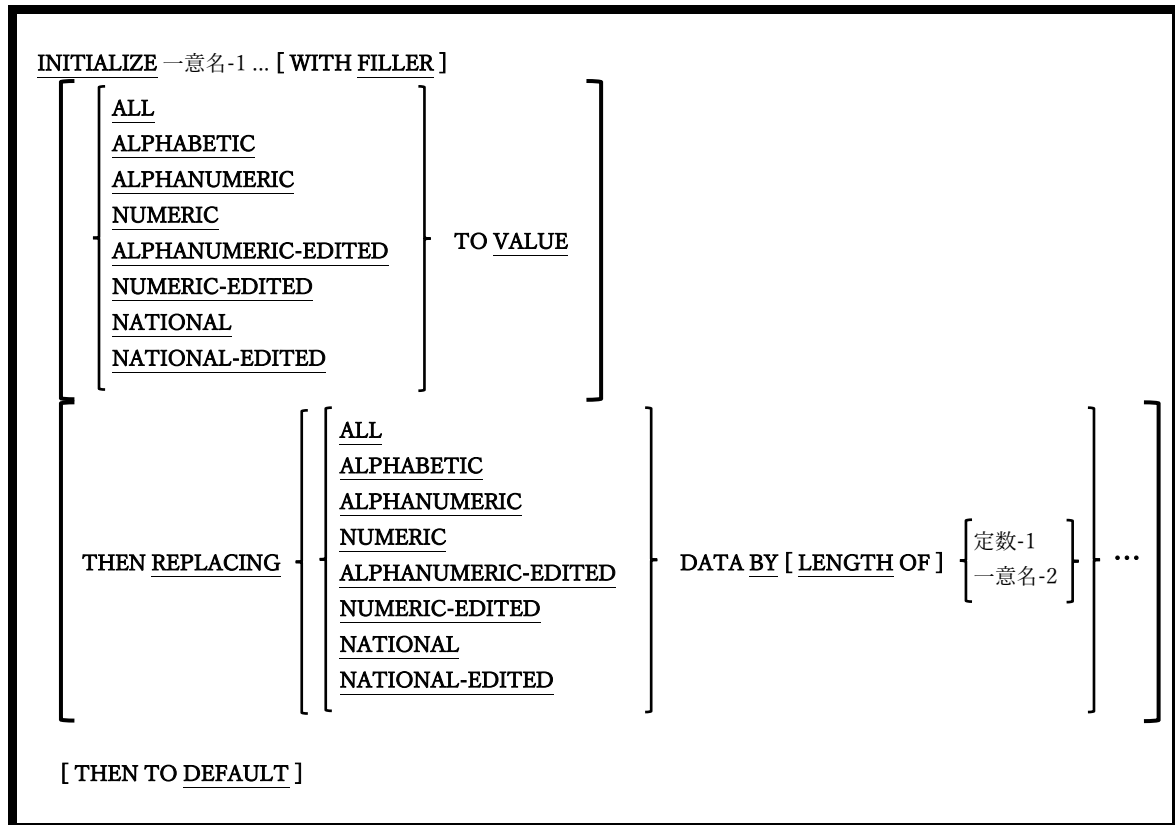
```
IF 条件式 THEN
 命令文-1
[ELSE 命令文-2]
[END-IF]
```

IF 文は、一つの命令文を条件付きで実行するため、または条件式の TRUE / FALSE 値に基づいて二つある命令文のうち一つを選択するために使われる。

1. 条件式が TRUE と評価された場合、ELSE 句が存在するかどうかに関係なく、命令文-1 が実行される。命令文-1 が実行されると、制御は END-IF 句に続く最初の文、END-IF 句がない場合は命令文に続く最初の文に移る。
2. ELSE 句が存在し、条件式-1 が FALSE と評価された場合、(その場合にのみ)命令文-2 が実行される。命令文-2 が実行されると、制御は END-IF 句に続く最初の文、END-IF 句がない場合は命令文に続く最初の文に移る。
3. ピリオド(.)と END-IF 文について、IF 文の範囲を終了できる方法が互いにどのように類似しているか、または異なっているかを、6.1.5 で例を挙げて説明している。

## 6.24. INITIALIZE

図 6-64-INITIALIZE 構文



INITIALIZE 文は、一意名-1 として指定された基本項目、または一意名-1 として指定された集団項目に従属する基本項目を特定の値に設定する。

1. これによって新しい値に設定できるデータ項目のリストは次の通りである。

- 一意名-1 として指定されたすべての基本項目。
- 一意名-1 として指定され、集団項目に従属して定義されたすべての基本項目。以下の例外を除く：
  - USAGE INDEX 項目は除外される。
  - 定義の一部として REDEFINES 句が含まれる項目は除外され、これに従属する



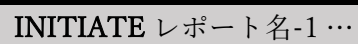
項目も除外される。ただし、一意名-1 の項目自体に REDEFINES 句が含まれている場合や、REDEFINES 句を含む項目に従属している場合がある。

以上は受け取り項目のリストである。

2. 一意名-1 項目の定義内、また、一意名-1 項目に従属する項目に OCCUR DEPENDING ON 句(5.3 参照)を含めることはできない。
3. オプションとして WITH FILLER 句が存在する場合、FILLER 項目は受け取り項目のリストに入る(そうでない場合は除外となる)。
4. TO VALUE 句または REPLACING 句が指定されていない場合、DEFAULT 句が指定されたとみなす。
5. オプションとして REPLACING 句が指定されている場合、INITIALIZE 文が構文的にコンパイラに受け入れられるためには、送信項目の MOVE 文が、すべての受け取り項目に対して有効でなければならない。
6. 各受け取り項目の初期化は、以下のルールに従って行われる。
  - TO VALUE 句が存在する場合、その受け取り項目は TO VALUE 句にリストされているデータカテゴリに含まれているか。含まれている場合、データ項目はその VALUE 句の値に初期化される。
  - REPLACING 句が存在する場合、その受け取り項目は REPLACING 句にリストされているデータカテゴリに含まれているか。含まれている場合、受け取り項目は指定された送信項目の値に初期化される。
  - DEFAULT 句が存在する場合は、項目値をその USAGE に適当な値に初期化する(英数字と数値は空白、ポインタとプログラムポインタは NULL、すべての数値と数値編集はゼロに初期化される)。

## 6.25. INITIATE

図 6-65-INITIATE 構文

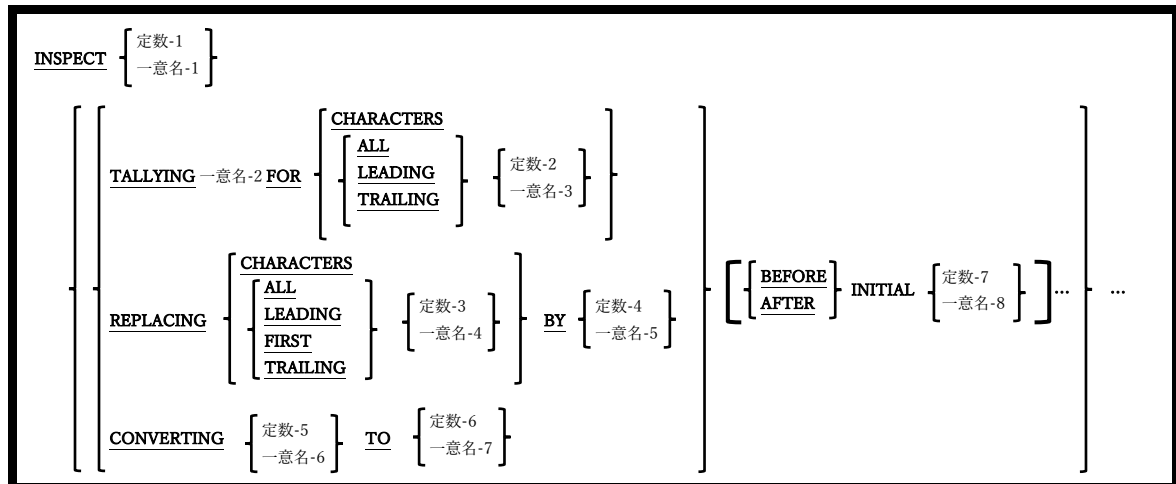


INITIATE レポート名-1 ...

INITIATE 文は、opensource COBOL コンパイラによって構文的には認識されるが、RWCS(COBOL Report Writer)は現在 opensource COBOL でサポートされていないため、機能しない。

## 6.26. INSPECT

図 6-66-INSPECT 構文



INSPECT 文は、文字列に対してさまざまなカウントまたはデータ変更操作を実行するために使われる。

- 一意名-1 および定数-1 は、英数字の USAGE DISPLAY データとして明示的または暗黙的に定義する必要があり、この時一意名-1 は集団項目の可能性はある。
- 定数-1 を指定すると、REPLACING 句または CONVERTING 句が使用できなくなる。
- 混同や衝突を避けるために、TALLYING、REPLACING、および CONVERTING 句は、コーディングされた順番で実行される。

INSPECT 文のルールは、指定された句によって異なる。

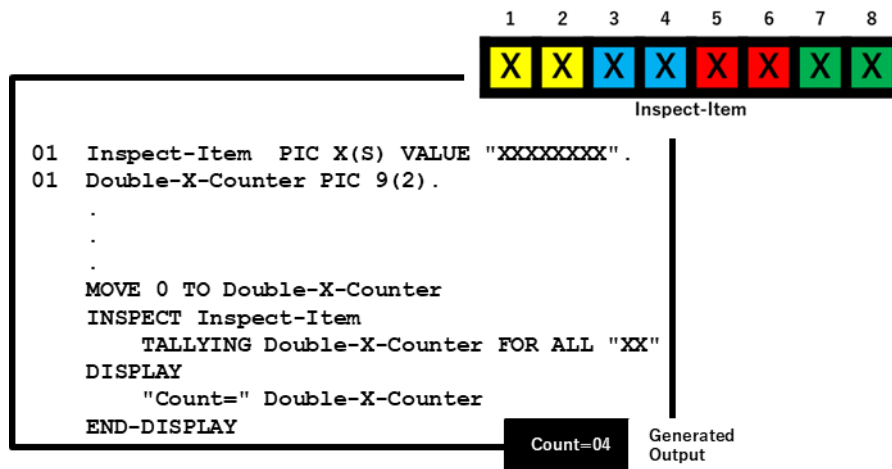
### TALLYING 句の場合：

TALLYING 句は、一意名-1 または定数-1 内の文字列数をカウントするために用いられる。

- 一意名-2 は編集不可の数値項目でなければならない。

2. 一意名-3 および定数-2 は、英数字の USAGE DISPLAY データとして明示的または暗黙的に定義する必要がある、この時一意名-3 は集団項目の可能性もある。
3. 一意名-2 は検索対象のターゲット文字列が一意名-1 で見つかるたびに、1 ずつ増加する。ターゲット文字列は以下のようになる。
  - a. CHARACTERS オプションが使用されている場合は 1 文字。基本的に合計文字数をカウントする。ALL、すべての LEADING、FIRST のみまたはすべての TRAILING の一意名-4 または定数-3 のオカレンス。
  - b. ALL、すべての LEADING、FIRST のみまたはすべての TRAILING の一意名-3 または定数-2 のオカレンス。
4. 通常は、定数-1 または一意名-1 の文字列全体がスキャンされる。ただし、この動作はオプションの BEFORE | AFTER 句を用いて変更することができ、スキャン対象の文字列で見つかったデータに基づいて開始点や終了点を指定できる。
5. ターゲット文字列が検出されて一致すると、INSPECT TALLYING プロセスは検出された文字列の最後から再開される。これにより、対象の文字列を重複してカウントしてしまうことを防ぐことができる。右の例は、「XX」オカレンスを検索する INSPECT TALLYING のオブジェクトとして使われる値が「XXXXXXXX」である 8 文字の項目を示す。

図 6-67-INSPECT 文 TALLYING 句の例



結果として、4 つの「XX」オカレンスのみが見つかりました。文字位置 2-3、4-5、および 6-7 も「XX」オカレンスではあるが、他のオカレンスと重複しているためカウントされない。

### REPLACING 句の場合：

REPLACING 句は、文字列内の部分文字列を、同じ長さで内容の異なるものに置き換えるために用いられる。1 つ以上の部分文字列を、長さも内容も異なる他の部分文字列に置き換える必要がある場合は、SUBSTITUTE 組み込み関数(6.1.7 参照)を使用すると良い。

- 一意名-4 および定数-3 は、英数字の USAGE DISPLAY データとして明示的または暗黙的に定義する必要があり、この時一意名-4 は集団項目の可能性はある。
- 一意名-5 および定数-4 は、英数字の USAGE DISPLAY データとして明示的または暗黙的に定義する必要があり、この時一意名-5 は集団項目の可能性はある。
- 一意名-4 と定数-3、一意名-5 と定数-4 は同じ長さでなければならない。
- 「BY」の前に指定された部分文字列は、ターゲット文字列と呼ばれ、「BY」の後

に指定された部分文字列は、置換文字列と呼ばれる。

6. ターゲット文字列は次のように識別できる：
  - a. CHARACTERS オプションが使用されている場合は、置換文字列の長さと同じ文字順序。
  - b. ALL、すべての LEADING、FIRST のみまたはすべての TRAILING の一意名-4 または定数-3 のオカレンス。
7. 通常は、一意名-1 の文字列全体がスキャンされる。ただし、この動作はオプションの BEFORE | AFTER 句を用いて変更することができ、スキャン対象の文字列で見つかったデータに基づいて開始点や終了点を指定できる。
8. ターゲット文字列が検出されて置き換えられると、INSPECT REPLACING プロセスは検出された文字列の最後から再開される。これにより、対象の文字列を重複して置き換えてしまうことを防ぐことができ、TALLYING の場合と非常に似ている。

#### CONVERTING 句の場合：

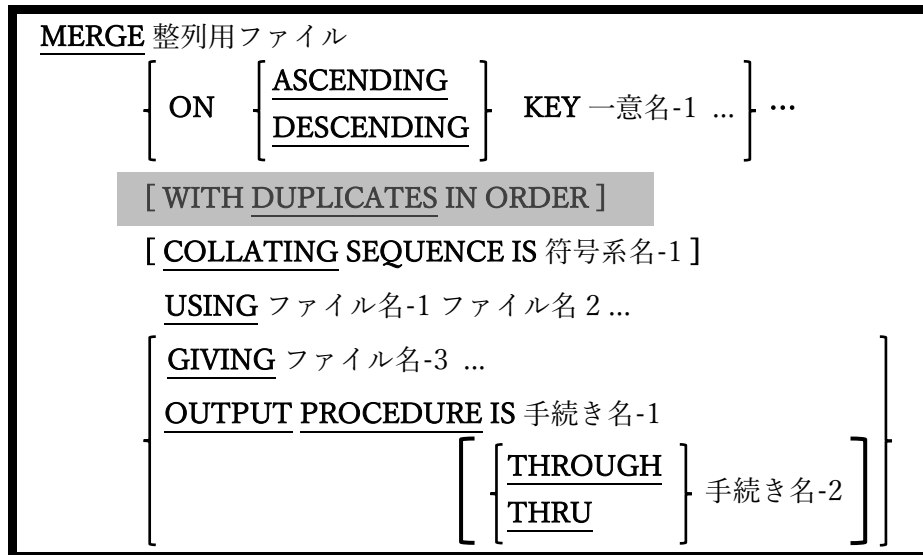
CONVERTING 句は、データ項目に対して単アルファベット置換を実行するために用いられる。

1. 一意名-5 および定数-6 は、英数字の USAGE DISPLAY データとして明示的または暗黙的に定義する必要がある、この時一意名-5 は集団項目の可能性はある。
2. 一意名-6 および定数-7 は、英数字の USAGE DISPLAY データとして明示的または暗黙的に定義する必要がある、この時一意名-6 は集団項目の可能性はある。
3. 一意名-5 と定数-6、一意名-6 と定数-7 は同じ長さでなければならない。

4. 「TO」の前に指定された部分文字列は、ターゲット文字列と呼ばれ、「TO」の後に指定された部分文字列は、置換文字列と呼ばれる。
5. 一意名-1 の内容は 1 文字ずつスキャンされ、その文字がターゲット文字列に該当する場合、(相対位置による)置換文字列内に対応する文字が、一意名-1 のその文字を置換する。
6. 置換文字列の長さがターゲット文字列の長さを超える場合、超過分は無視される。
7. ターゲット文字列の長さが置換文字列の長さを超える場合、置換文字列の右側に空白があると見なされてその差が埋められる。
8. INSPECT 文は 1985 年の COBOL 標準で導入されたため、TRANSFORM 文(6.47)は廃止された。

## 6.27. MERGE

図 6-68-MERGE 構文



MERGE 文は、指定されたキーのセットで二つ以上の同じ順序ファイルを結合する。

1. MERGE 文で指定された整列ファイルは、データ部のファイル節でソート記述(SD)を使って定義する必要がある。5.2 では説明の残りの部分で、このファイルを「マージファイル」と呼んでいる。
2. ファイル名-1、ファイル名-2、およびファイル名-3(指定されている場合)は、ORGANIZATION LINE SEQUENTIAL または ORGANIZATION RECORD BINARY SEQUENTIAL ファイルを参照する必要がある。これらのファイルは、データ部のファイル節でファイル記述(FD)を使って定義しなければならない。5.1 ではファイル名-1 とファイル名-2 で同じファイルが使われている。
3. 一意名-1…の項目は、整列ファイルのレコード内の項目として定義する必要がある。
4. WITH DUPLICATES IN ORDER 句は互換性のためにサポートされているが機能していない。
5. ファイル名-1、ファイル名-2、ファイル名-3(存在する場合)、および整列ファイルのレ



コード記述は、レイアウトとサイズが同じであると見なされる。ファイルレコードの項目に使われる実際のデータ名は異なる場合があるが、レコードの構造、項目の PICTURE 句、項目のサイズ、およびデータの USAGE 句は、すべてのファイルで項目ごとに一致する必要がある。

MERGE 文を使った一般的なプログラミング手法は、MERGE に関連するすべてのファイルのレコードを、「01 レコード名 PIC X(n).」(n はレコードサイズを表す)という書き方の簡潔な基本項目として定義することである。レコードの詳細が実際に記述されている唯一のファイルが整列ファイルである。

6. USING 句で指定されたファイルには、以下のルールが適用される。
  - a. MERGE の実行時は、いずれのファイルも OPEN になっていない場合がある。
  - b. 各ファイルは、MERGE 文の KEY 句での指定によって既に並び替えられているとみなされる。
  - c. SAME RECORD AREA、SAME SORT AREA、または SAME SORT-MERGE AREA 文で参照できるファイルはない<sup>21</sup>。
7. MERGE を実行すると、各 USING ファイルの最初のレコードが読み取られる。
8. MERGE 文が実行されると、各 USING ファイルの現在のレコードが調査され、KEY 句によって規定されたルールに沿って比較される。(KEY 句による)順番で見て「次」であるレコードがマージファイルに書き込まれると、そのレコードの元となった USING ファイルが読み取られて、次の順番のレコードが使用できるようになる。USING ファイルがファイル終了条件に達すると、そのファイルはそれ以降の MERGE 処理から除外され、処理は残りの USING ファイルで続行される。すべての USING ファイルでの処理が完全に終わるまで続く。

---

<sup>21</sup> 4.2.2 参照。

9. マージファイルにデータが入力されると、GIVING 句が指定されている場合、マージされたデータはファイル名-3 に書き込まれるか、手続き名-1 または 手続き名-1 と 手続き名-2 の間として定義されている OUTPUT PROCEDURE を使って処理される。
10. GIVING を指定する場合、MERGE の実行時にファイル名-3…を OPEN にすることはできない。
11. OUTPUT PROCEDURE を使用する場合、マージされたレコードは RETURN 文(6.35) を用いて、マージファイルから一つずつ手動で読み取られる。
12. OUTPUT PROCEDURE 内で実行された STOP RUN、EXIT PROGRAM、または GOBACK は、現在実行中のプログラムと MERGE 文を終了する。
13. OUTPUT PROCEDURE から制御を移した GO TO 文は MERGE を終了するが、GO TO 文が制御を移した場所からプログラムの実行を継続できるようにする。GO TO を用いて OUTPUT PROCEDURE を中止してしまうと、再開することはできないが、MERGE 文自体は再び実行することができる。しかし、この方法で MERGE を再起動すると、マージファイルから返されていないレコードは失われてしまう。**GO TO を使用することで並び替えを早期に終了したり、以前に中止された MERGE を再開したりすることは、優れたプログラミング方法ではないため、避けるべきである。**
14. OUTPUT PROCEDURE は、手続き名-2(該当するものがない場合は手続き名-1)の最後の文を過ぎた制御のフォールスルーによって暗黙的に終了するか、手続き名-2(該当するものがない場合は手続き名-1)で実行される EXIT SECTION / EXIT PARAGRAPH を介して明示的に終了する。OUTPUT PROCEDURE が終了すると、出力フェーズ(および MERGE 文自体)が終了となる。
15. OUTPUT PROCEDURE の範囲では、ファイルの SORT 文(6.40.1)、MERGE 文、または RELEASE 文(6.34)を実行してはならない。

## 6.28. MOVE

### 6.28.1. MOVE 文の書き方 1 — MOVE

図 6-69-MOVE 構文

|                                                                                                           |
|-----------------------------------------------------------------------------------------------------------|
| <u>MOVE</u> $\left[ \begin{array}{l} \text{定数-1} \\ \text{一意名-1} \end{array} \right]$ <u>TO</u> 一意名-2 ... |
|-----------------------------------------------------------------------------------------------------------|

特定の値を一つ以上の受け取りデータ項目に移動することができる。

1. MOVE 文は、一つ以上の受け取りデータ項目(一意名-2...)の内容を新しい値に置き換える。
2. 新しい値が各受け取りデータ項目に格納される正確な方法は、各一意名-2 項目の PICTURE と USAGE によって異なる。

### 6.28.2. MOVE 文の書き方 2 — MOVE CORRESPONDING

図 6-70-MOVE CORRESPONDING 構文

|                                                     |
|-----------------------------------------------------|
| <u>MOVE CORRESPONDING</u> 一意名-1 <u>TO</u> 一意名-2 ... |
|-----------------------------------------------------|

同じ名前の基本項目をある集団項目から別の集団項目に移動することができる。

1. CORRESPONDING という単語は、CORR と省略される場合がある。
2. 一意名-1 と一意名-2 の両方が集団項目でなければならない。
3. 一意名-1 と一意名-2 に従属する二つのデータ項目は、次の条件を満たす場合に対応すると言われている：
  - a. どちらも同じ名前ではあるが FILLER ではない。

- b. 一意名-1 と一意名-2 に直ちには従属しない場合、上位項目は同じ名前ではあるが FILLER ではない。これらの項目が一意名-1 と一意名-2 でない場合、このルールは一意名-1 と一意名-2 の構造を通じて再帰的に上位の方に適用されていく。
  - c. どちらも基本項目(ADD CORR、SUBTRACT CORR)であるか、少なくとも一つが基本項目(MOVE CORR)である。
  - d. 対応する可能性のある候補は、別のデータ項目の REDEFINES 句または RENAMES 句ではない。
  - e. 対応する可能性のある候補のいずれにも OCCURS 句はない(ただし OCCURS 句を含む従属データ項目が含まれている場合がある)。
4. 対応するものとの一致が確認できると、MOVE CORRESPONDING は合致すること  
に一つずつ、個々に MOVE が行われたかのように動作する。

この規則は、以下の例題を使うとよく理解できる。

```
IDENTIFICATION DIVISION.
PROGRAM-ID. corrdemo.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X.
 05 A VALUE 'A' PIC X(1) .
 05 G1.
 10 G2.
 15 B VALUE 'B' PIC X(1) .
 05 C.
 10 FILLER VALUE 'C' PIC X(1) .
 05 G3.
 10 G4.
 15 D VALUE 'D' PIC X(1) .
 05 V1 VALUE 'E' PIC X(1) .
 05 E REDEFINES V1 PIC X(1) .
 05 F VALUE 'F' PIC X(1) .
 05 G VALUE ALL 'G' .
 10 G2 OCCURS 4 TIMES PIC X(1) .
 05 H VALUE ALL 'H' PIC X(4) .
```

```
01 Y.
 02 A PIC X(1) .
 02 G1.
 03 G2.
 04 B PIC X(1) .
 02 C PIC X(1) .
 02 G3.
 03 G5.
 04 D PIC X(1) .
 02 E PIC X(1) .
 02 V2 PIC X(1) .
 02 G PIC X(4) .
 02 H OCCURS 4 TIMES PIC X(1) .
 66 F RENAMEs V2.
PROCEDURE DIVISION.
100-Main.
 MOVE ALL '-' TO Y.
 DISPLAY ' Names: ' 'ABCDEFGGGGHHHH'.
 DISPLAY 'Before: ' Y.
 MOVE CORR X TO Y.
 DISPLAY ' After: ' Y.
 STOP RUN.
```

DISPLAY 文で表示される結果は以下の通りである。

```
Names: ABCDEFGGGGHHHH
Before: -----
After: ABC---GGGG----
```

- opensource COBOL では、「X」および「Y」集団項目内の「A」、「B」、および「C」データ項目間の「対応する」関係を確認している。「X」は 01-05-10-15 のレベル番号付けスキームを使用し、「Y」は 01-02-03-04 を使用しているが、この違いは対応するものの一致が確立することに影響しない。
- G OF X は OCCURS 句を含むデータ項目の親であるが、「G」項目が一致する。
- 「D」項目は 3 項の b に違反しているため、一致するものはない(4 つの集団項目名を注視すること)。
- E OF X は 3 項の d(REDEFINES)に違反しているため、「E」項目と一致するものはない。
- E OF X は 3 項の d(RENAMEs)に違反しているため、「F」項目と一致するものはない。

- H OF Y には OCCURS 句が含まれており、3 項の e に違反しているため、「H」項目と一致するものはない。

## 6.29. MULTIPLY

### 6.29.1. MULTIPLY 文の書き方 1 — MULTIPLY BY

図 6-71-MULTIPLY BY 構文

```
MULTIPLY { 定数-1
 一意名-1 } BY { 一意名-2 [ROUNDED] } ...

[ON SIZE ERROR 命令文-1]

[NOT ON SIZE ERROR 命令文-2]

[END-MULTIPLY]
```

算術積を実行する。

1. 一意名-1 および一意名-2 は、編集不可の数値データ項目でなければならない。
2. 定数-1 は数字定数でなければならない。
3. それぞれ一意名-2 を掛けた一意名-1 または integer-1 の値が計算され、各計算結果が対応する一意名-2 データ項目に移動され、古い内容が置き換えられる。
4. ON SIZE ERROR、NOT ON SIZE ERROR、および ROUNDED 句はコード化され、ADD 文での同名義句と同様に動作する(6.5 参照)。

### 6.29.2. MULTIPLY 文の書き方 2 — MULTIPLY GIVING

図 6-72-MULTIPLY GIVING 構文

```
MULTIPLY { 定数-1
 一意名-1 } BY { 定数-2
 一意名-2 }
GIVING { 一意名-3 [ROUNDED] } ...

[ON SIZE ERROR 命令文-1]

[NOT ON SIZE ERROR 命令文-2]

[END-MULTIPLY]
```

二つの値の算術積を実行し、GIVING の後にリストされている一意名(一意名-3...)の内容をその積に置き換える。

1. 一意名-1 および一意名-2 は、編集不可の数値データ項目でなければならない。
2. 一意名-3 は数値データ項目でなければならないが、編集可能な場合もある。
3. 定数-1 と定数-2 は数字定数でなければならない。
4. 一意名-1 および一意名-2 の値は変更できない。
5. ON SIZE ERROR、NOT ON SIZE ERROR、および ROUNDED 句はコード化され、ADD 文での同名義句と同様に動作する(6.5 参照)。



## 6.30. NEXT SENTENCE

図 6-73-NEXT SENTENCE 構文



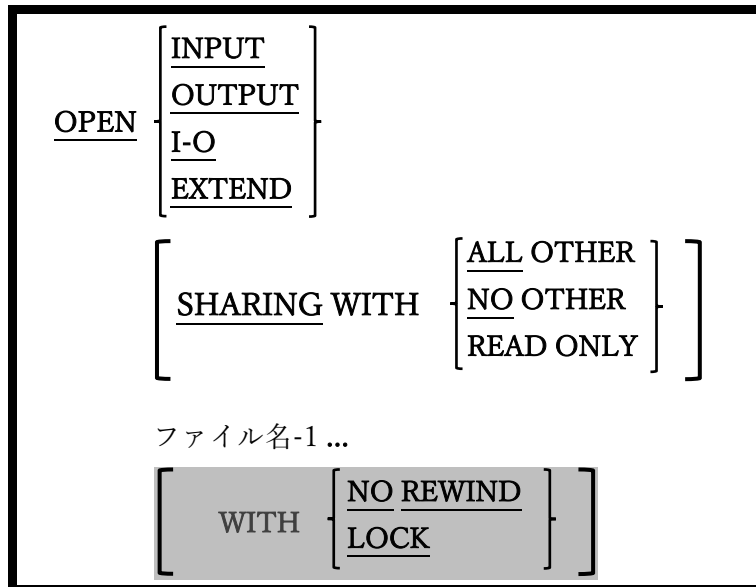
NEXT SENTENCE

NEXT SENTENCE 文は、ネストされた一連の「IF」文を「分割」する手段として使われる。

1. NEXT SENTENCE 文は、「IF」文内で使用する場合にのみ有効である。
2. 名前が示すように、この文によって制御はプログラム内の次の文に移る。
3. 1985 年より前の標準に従ってコーディングされた COBOL プログラムに NEXT SENTENCE 文が必要な理由については、6.1.5 で説明している。また、1985 年(およびそれ以降)の標準用にコーディングされたプログラムがこの文を必要としない理由もわかるだろう。
4. 新しい opensource COBOL プログラムは、IF 文に END-IF スコープターミネータを使ってコーディングする必要がある。これにより、CONTINUE 文(6.12)を優先することで NEXT SENTENCE の使用が無効となる。

## 6.31. OPEN

図 6-74-OPEN 構文



OPEN 文は、プログラム内の一つ以上のファイルを使用できるようにする。

1. opensource COBOL プログラムで定義されたファイルは、CLOSE 文(6.9)、DELETE 文(6.13)、READ 文(6.33)、START 文(6.41)、または UNLOCK 文(6.48)で参照される前に、正常に OPEN されている必要がある。更に、ファイルのレコードデータ名 (またはレコードに従属するデータ要素)を ANY 文で参照するためには、ファイルが正常に OPEN されていなければならない。
2. 既に開いているファイルを開こうとすると、ファイルステータス 41(「ファイルは既に開いています」)で失敗となり、これはプログラムを終了させてしまう致命的なエラーとなる。
3. OPEN の失敗(「ファイルは既に開いています」を含む)は、DECLARATIVES(6.3)またはエラープロシージャ(7.3.2)を使って処理できるが、トラップルーチンが終了してしまうと、opensource COBOL ランタイムシステムはプログラムを終了し、最終的に OPEN 障害から回復することはできない。

|                                    |      |
|------------------------------------|------|
| opensource COBOL Programmers Guide | 手続き部 |
|------------------------------------|------|

4. INPUT、OUTPUT、I-O、および EXTEND オプションは次のように、ファイルの使用方法を opensource COBOL に通知する。

| オプション  | 処理                                                                             |
|--------|--------------------------------------------------------------------------------|
| INPUT  | ファイルの既存内容のみを読み取ることができ、CLOSE、READ、START、および UNLOCK 文のみが許可される。                   |
| OUTPUT | 新しい内容(ファイルの既存内容が完全に置き換わる場合)のみをファイルに書き込むことができ、CLOSE、UNLOCK、および WRITE 文のみが許可される。 |
| I-O    | ファイルに対して任意の操作を実行でき、すべてのファイル操作 I/O 文が許可される。                                     |
| EXTEND | 新しい内容(ファイルの既存内容に追加される場合)のみをファイルに書き込むことができ、CLOSE、UNLOCK、および WRITE 文のみが許可される。    |

5. SHARING 句は、同じファイルを開こうとする他の opensource COBOL プログラムと自分のプログラムがどのように共存するかを opensource COBOL に通知する。このオプションについては 6.1.9.1 で説明している。
6. WITH NO REWIND 句と WITH LOCK 句は機能しない。

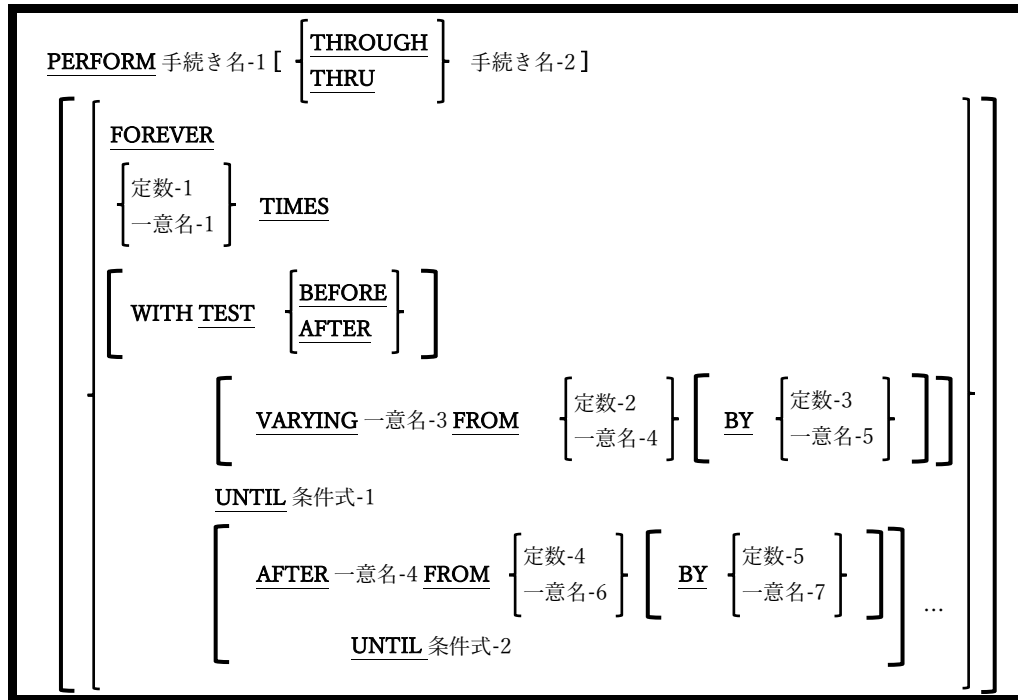
WITH NO REWIND 句をサポートできるデバイス(テープドライブ)は、opensource COBOL が動作する環境では非常に稀であり、コンパイラまたはランタイムメッセージは発行されない(何も実行されない)。

WITH LOCK 句は正式には「実装されていない」ため扱いが少し異なり、コンパイル警告が生成される。

## 6.32. PERFORM

### 6.32.1. PERFORM 文の書き方 1 — 手続き型

図 6-75-手続き型 PERFORM 構文



制御を一つ以上のプロシージャに移し、指定されたプロシージャの実行が完了したときに制御を返すために使われる。このプロシージャの呼び出しは、条件が TRUE になるまで、または永久に(おそらくプロシージャ内の PERFORM の制御から抜け出す方法で)、一回、複数回、繰り返し実行できる。

1. THROUGH と THRU の単語は、同じ意味を持つものとして使用することができる。
2. 手続き名-1 と手続き名-2 はどちらも、PERFORM 文と同じプログラム単位で定義された手続き部の節または段落でなければならない。
3. 手続き名-2 オプションを指定する場合は、プログラムのソースコード内にある手続き名-1 に従う必要がある。

4. PERFORM の範囲は、手続き名-1 内の文、手続き名-2 内の文、およびこれらの間で定義された全プロシージャ内のすべての文として定義される。
5. FOREVER、TIMES、または UNTIL 句が存在しない場合、PERFORM の範囲内のコードが(一度)実行された後、制御は PERFORM に続く文に移る。
6. FOREVER オプションは、PERFORM 文に繰り返しの終了条件が定義されていない場合、PERFORM の範囲内でコードを繰り返し実行する。プログラムを停止する(STOP RUN)か、PERFORM から抜け出す(EXIT PERFORM)コードを PERFORM の範囲内に含めるのかどうかは、プログラマ次第である。
7. TIMES オプションは、PERFORM の範囲内で一定回数、指示された実行を繰り返す。指定された回数分の繰り返しが終了すると、制御は PERFORM に続く次の文に移る。
8. UNTIL 句を用いると、PERFORM の範囲内の文を、条件式-1 の値が TRUE になるまで繰り返し実行できる。
9. オプションの WITH TEST 句は UNTIL が、PERFORM 範囲の前に実行されるか、後に実行されるかを制御する。WITH TEST 句が指定されていない場合は「BEFORE」が指定されたものとみなす。
10. オプションの VARYING 句を使うと、PERFORM の範囲内で文を実行するたびに一意の数値を持つデータ項目(一意名-3)を定義できる。初め一意名-3 は FROM 句で指定された値を持つ。反復の終了時に、BY 句で定義された値は、条件式-1 が評価される前に一意名-3 に追加される。BY 句が指定されていない場合は「1」が指定されたものとみなす。
11. VARYING 句が使用されている場合は、任意の数だけ AFTER 句を追加して、二次ループを作成することができる。AFTER 句では反復を追加作成し、反復中に増加する追加のデータ項目を定義し、反復を終了するために追加の条件式を定義することができる。

る。機能的には、複数の文をコーディングすることなく、ある PERFORM / VARYING / UNTIL を別の PERFORM / VARYING / UNTIL 内にネストする基本的な方法である。次の例が参考になるだろう。

2次元(3行×4列)のテーブルと、テーブルの各要素への添字参照に使用される数値データ項目のペ  
アを定義する次のコードを確認する。

|           |           |           |           |
|-----------|-----------|-----------|-----------|
| PD (1, 1) | PD (1, 2) | PD (1, 3) | PD (1, 4) |
| PD (2, 1) | PD (2, 2) | PD (2, 3) | PD (2, 4) |
| PD (3, 1) | PD (3, 2) | PD (3, 3) | PD (3, 4) |

```

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.

```

ルーチン(100-Visit-Each-PD)を PERFORM したいとする。このルーチンは、右側に示した順序で各 PD データ項目に順次アクセスする。  
PERFORM コードは次の通りである。

|   |    |    |    |
|---|----|----|----|
| 1 | 2  | 3  | 4  |
| 5 | 6  | 7  | 8  |
| 9 | 10 | 11 | 12 |

```

PERFORM 100-Visit-Each-PD 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.

```

|   |   |   |    |
|---|---|---|----|
| 1 | 4 | 7 | 10 |
| 2 | 5 | 8 | 11 |
| 3 | 6 | 9 | 12 |

一方で左に示した順序で各 PD にアクセスしたい場合、必要な PERFORM コードは次の通りである。

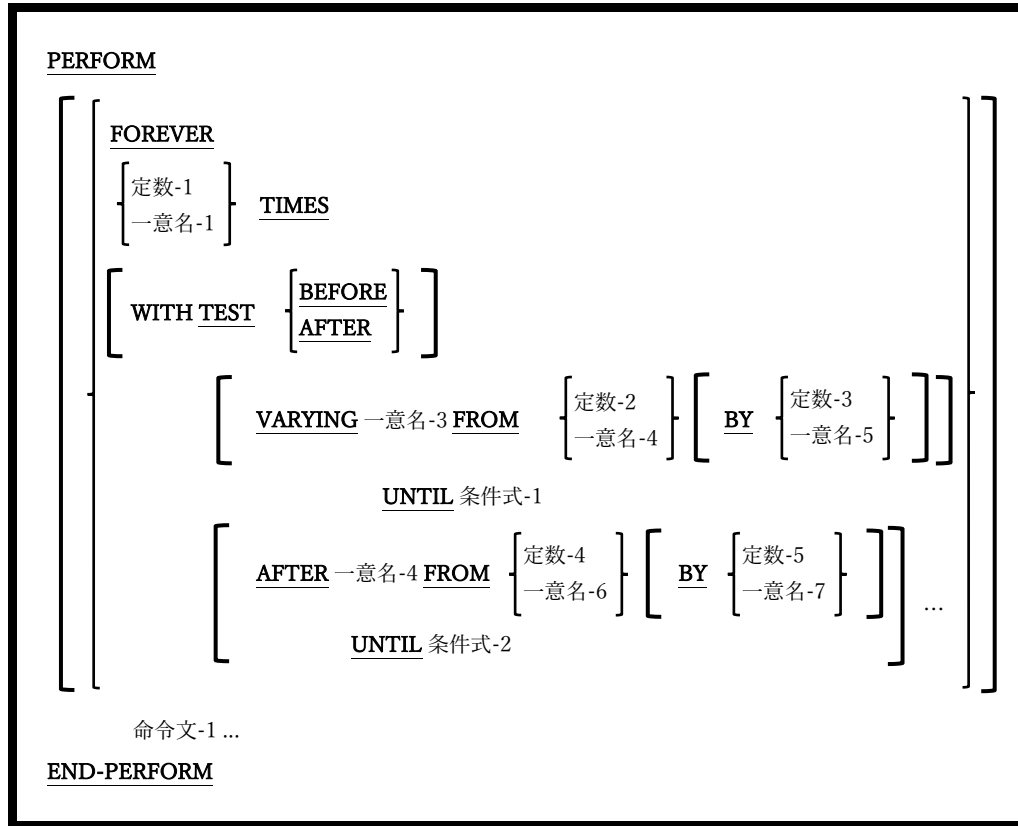
```

PERFORM 100-Visit-Each-PD WITH TEST AFTER
 VARYING PD-Col-No FROM 1 BY 1 UNTIL PD-Col-No = 4
 VARYING PD-Row-No FROM 1 BY 1 UNTIL PD-Row-No = 3.

```

### 6.32.2. PERFORM 文の書き方 2 — インライン型

図 6-76-インライン型 PERFORM 構文



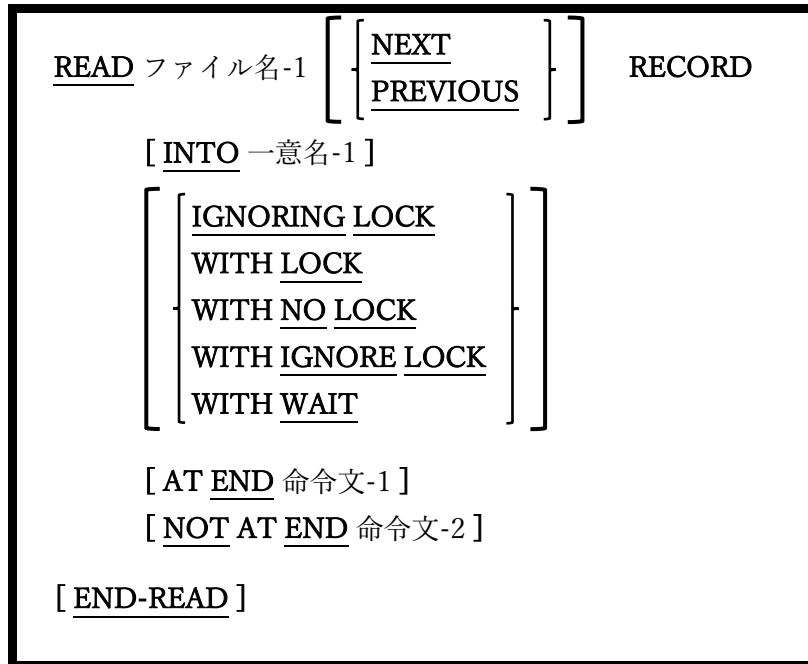
PERFORM の範囲内にある文が、プログラム内の他の場所にあるプロシージャではなく、PERFORM のコードにインラインで指定されること以外は、書き方 1 と同じである。

1. FOREVER、TIMES、WITH TEST、VARYING、BY、AFTER、および UNTIL 句は、PERFORM 文の書き方 1 の同名義句と、使い方や効果が同じである。
2. この書き方と書き方 1 の明確な違いは、書き方 2 の PERFORM 文では、実行コードがプロシージャではなくインライン(命令文 1...)で指定されることである。

## 6.33. READ

### 6.33.1. READ 文の書き方 1 — 順次読み取り

図 6-77-READ 構文(順次読み取り)



ファイルから次の(または前の)レコードを取得する。

1. ファイル名-1 は、INPUT または I-O に対して常に OPEN(6.31)である必要がある。
2. ファイル名-1 の ACCESS MODE が RANDOM の場合、この書き方の READ 文は使用できない。
3. ACCESS MODE が SEQUENTIAL の場合、この書き方の READ 文が唯一使用可能となり、NEXT / PRIOR 句はオプションとして扱われる。
4. ACCESS MODE が DYNAMIC の場合、書き方 2 と同様にこの書き方の READ 文も使用できる。以下、最小限の READ 文は…

READ ファイル名-1

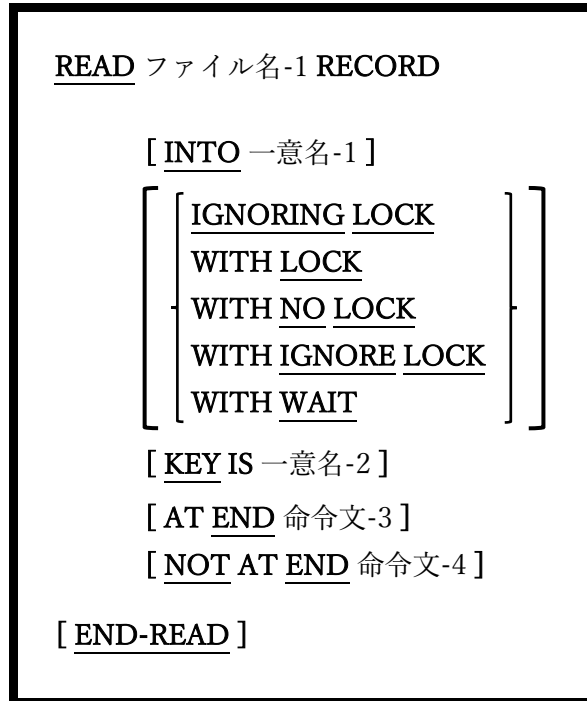


…正しい書き方として認められる。そのため、ACCESS MODE DYNAMIC が指定されていて、上記のような文を順次読み取りとして処理することを opensource COBOL コンパイラに通知する場合は、文に NEXT または PRIOR を追加する必要がある(そうでない場合は、ランダム読み取りとして扱われる)。

5. ファイル名-1 で次に使用可能なレコードが取得され、その内容はファイルの FD(5.1) に従属する 01 レベルのレコード構造に格納される。
6. NEXT 句および PREVIOUS 句では、読み取りプロセスがどの方向でファイルを通過するかを指定する。どちらも指定されていない場合は、NEXT が指定されているものとみなされる。
7. PREVIOUS 句は、ORGANIZATION INDEXED ファイルでのみ使うことができる。
8. INTO 句を使うと、読み取りが成功した場合、読み取ったレコード内容が MOVE の規則に従って一意名-1 に MOVE される。
9. レコードの LOCK 句については 6.1.9.2 で説明している。
10. AT END 句が存在する場合、ファイルステータスが 10「ファイルの終わり」であることが原因で READ の試行が失敗した時に命令文 1 を実行する。AT END 句は、**ゼロ以外のファイルステータス値を検出しないため**、DECLARATIVES ルーチン(6.3)または READ 後に明示的に宣言されたファイルステータス項目を使って、ファイルの終わり以外のエラー状態を検出する。
11. NOT AT END 句が存在する場合、READ の試行が成功すると、命令文 2 が実行される。

### 6.33.2. READ 文の書き方 2 — ランダム読み取り

図 6-78-READ 構文(ランダム読み取り)



ファイルから任意のレコードを取得する。

1. ファイル名-1 は、INPUT または I-O に対して常に OPEN(6.31)である必要がある。
2. ファイル名-1 の ACCESS MODE が SEQUENTIAL,の場合、この書き方の READ 文は使用できない。
3. ACCESS MODE が RANDOM の場合、この書き方の READ 文が唯一使用可能となる。
4. ACCESS MODE が DYNAMIC の場合、書き方 2 と同様にこの書き方の READ 文も使用できる。以下、最小限の READ 文は…

READ ファイル名-1

…正しい書き方として認められる。そのため、ファイルに ACCESS MODE DYNAMIC が指定されている場合、上記のような READ 文は自動的にランダム読み取りとして扱われる。

5. KEY 句は、ファイル内でレコードをどのように配置するかをコンパイラに指示する。

KEY 句がない場合：

- ファイルが ORGANIZATION RELATIVE ファイルの場合、ファイルの RELATIVE KEY として宣言された項目の内容がレコードの識別に使われる。
- ファイルが ORGANIZATION INDEXED ファイルの場合、ファイルの RECORD KEY として宣言された項目の内容がレコードの識別に使われる。

KEY 句が指定されている場合：

- ファイルが ORGANIZATION RELATIVE ファイルの場合、一意名-2 の内容が、アクセスされるレコードの相対レコード番号として使われる。一意名-2 は、ファイルの RELATIVE KEY 項目である必要はない(必要に応じて指定することが可能)。
  - ファイルが ORGANIZATION INDEXED ファイルの場合、一意名-2 は RECORD KEY またはファイルの ALTERNATE RECORD KEY 項目の一つ(存在する場合)である必要があり、その項目の最新の内容によって、アクセスするレコードが識別される。代替レコードキーが使用され、重複値が許可されている場合、アクセスされるレコードは、そのキー値を持つ最初のレコードになる。
6. 5 項で識別されるレコードはファイル名-1 から取得され、その内容はファイルの FD(5.1)に従属する 01 レベルのレコード構造に格納される。
7. INTO 句を使うと、読み取りが成功した場合、読み取ったレコード内容が MOVE の規

則に従って一意名-1 に MOVE される。

8. レコードの LOCK 句については 6.1.9.2 で説明している。
9. INVALID KEY 句が存在する場合、ファイルステータスが 23「キーが存在しない」であることが原因で READ の試行が失敗した時に命令文 1 を実行する。INVALID KEY 句は、**ゼロ以外のファイルステータス値を検出しないため**、DECLARATIVES ルーチン(6.3)または READ 後に明示的に宣言されたファイルステータス項目を使って、「キーが存在しない」以外のエラー状態を検出する。
10. NOT INVALID KEY 句が存在する場合、READ の試行が成功すると、命令文 2 が実行される。

## 6.34. RELEASE

図 6-79-RELEASE 構文

|                                                                                                   |               |
|---------------------------------------------------------------------------------------------------|---------------|
| <u>RELEASE</u> レコード名-1 [ <u>FROM</u> <table border="1"><tr><td>定数-1<br/>一意名-1</td></tr></table> ] | 定数-1<br>一意名-1 |
| 定数-1<br>一意名-1                                                                                     |               |

RELEASE 文は、整列ファイルに新しいレコードを追加する。

1. RELEASE 文は、SORT 文の INPUT PROCEDURE 内でのみ有効である(6.40.1 参照)。
2. レコード名-1 は、ソート記述(SD)記述項に定義されたレコードでなければならない(5.2 参照)。

## 6.35. RETURN

図 6-80-RETURN 構文

RETURN ファイル名-1 RECORD

[ INTO 一意名-1 ]

[ AT END 命令文-1 ]

[ NOT AT END 命令文-2 ]

[ END-READ ]

RETURN 文は、整列ファイルまたはマージファイルからレコードを読み取る。

1. RETURN 文は、SORT 文(6.40.1)またはMERGE 文(6.27)の OUTPUT PROCEDURE 内でのみ有効である。
2. ファイル名-1 は、ソート記述(SD)記述項で定義された整列ファイルまたはマージファイルでなければならない(5.2 参照)。
3. INTO、AT END、および NOT AT END 句は、READ 文(6.33)と同様にして扱われる。

## 6.36. REWRITE

図 6-81-REWRITE 構文

REWRITE レコード名-1

[ FROM { 定数-1  
一意名-1 } ]

[ [ WITH LOCK  
WITH NO LOCK ] ]

[ INVALID KEY 命令文-3 ]

[ NOT INVALID KEY 命令文-4 ]

[ END-REWRITE ]

REWRITE 文は、ディスクファイル上の論理レコードを置き換える。

1. レコード名-1 は、I-O に対して現在 OPEN(6.31)になっているファイルのファイル記述(FD-5.1 参照)に従属する 01 レベルのレコードとして定義される必要がある。
2. FROM 句を使うと、レコード名-1 をファイルに書き込む前に、定数-1 または一意名-1 が暗黙的にレコード名-1 への MOVE が発生する。
3. REWRITE 文は、ORGANIZATION IS LINE SEQUENTIAL ファイルでは使用できない。
4. レコードの LOCK 句については 6.1.9.2 で説明している。
5. レコードを書き換えても、ファイルの次のブロックが読み取られるか、COMMIT 文(6.10)が発行されるか、そのファイルが閉じられるまで、ファイルのレコードの内容は物理的に更新されない。

6. ファイルに ORGANIZATION RECORD BINARY SEQUENTIAL がある場合：
  - a. 書き換えられるレコードは、ファイルの最後に実行された READ 文(6.33)によって取得されたレコードとなる。
  - b. レコード名-1 のサイズは変更できません (5.1 の RECORD CONTAINS / RECORD IS VARYING 句を参照)。
7. ファイルに ORGANIZATION RELATIVE または ORGANIZATION INDEXED がある場合：
  - a. ACCESS MODE SEQUENTIAL がある場合、書き換えられるレコードは、ファイルの最後に実行された READ 文(6.33)によって取得されたレコードとなる。ACCESS MODE RANDOM または ACCESS MODE DYNAMIC がある場合、レコードを書き換える前の READ 文は必要ない。ファイルの RELATIVE KEY / RECORD KEY 定義で、更新するレコードを指定する。
  - b. レコード名-1 のサイズは更新される可能性がある。
8. REWRITE 文の実行中にエラーが発生した場合、ON INVALID KEY 句が実行される (つまり命令文 1 が実行される)。このようなエラーは、実際の I/O エラーまたは「キーが存在しない」エラー(ファイルステータス 23)である可能性があり、RELATIVE KEY または RECORD KEY 句の要件を満たすレコードが存在しないことを示す。
9. REWRITE 文の実行中にエラーが発生しなかった場合、NOT ON INVALID KEY 句が実行され、命令文 2 が実行される。



## 6.37. ROLLBACK

図 6-82-ROLLBACK 構文

ROLLBACK

ROLLBACK 文は、プログラムの開始以降または最後の COMMIT 以降に行われたすべてのファイルへの変更を元に戻す。

1. opensource COBOL は(少なくとも今現在)ファイルのロールバックをサポートしていない。ROLLBACK 文は、COMMIT 文(6.10)と同じ働きをする。

## 6.38. SEARCH

### 6.38.1. SEARCH 文の書き方 1 — 順次探索

図 6-83-SEARCH 構文(順次探索)

```
SEARCH テーブル名
 [VARYING 指標名-1]
 [AT END 命令文-1]
 { WHEN 条件式-1 命令文-2 }...
 [END-SEARCH]
```

SEARCH 文は、テーブルを順に探索するために使われ、特定の値がテーブル内に配置されるか、テーブルが完全に探索されると停止する。

1. VARYING 句で指定された指標名-1 一意名は、USAGE INDEX でなければならない。
2. VARYING 句が指定されていない場合、探索対象のテーブルは INDEXED BY 句(5.3 を参照)を用いて作成する必要がある。
3. SEARCH 文の実行時に、指標名-1(またはテーブルで定義されている INDEXED BY 索引)の現在の値によって、探索プロセスを実行するテーブルの開始位置が定義される。通常は次の例のように、SEARCH 文を開始する前に索引値を 1 に初期化する：

SET 指標名-1 TO 1

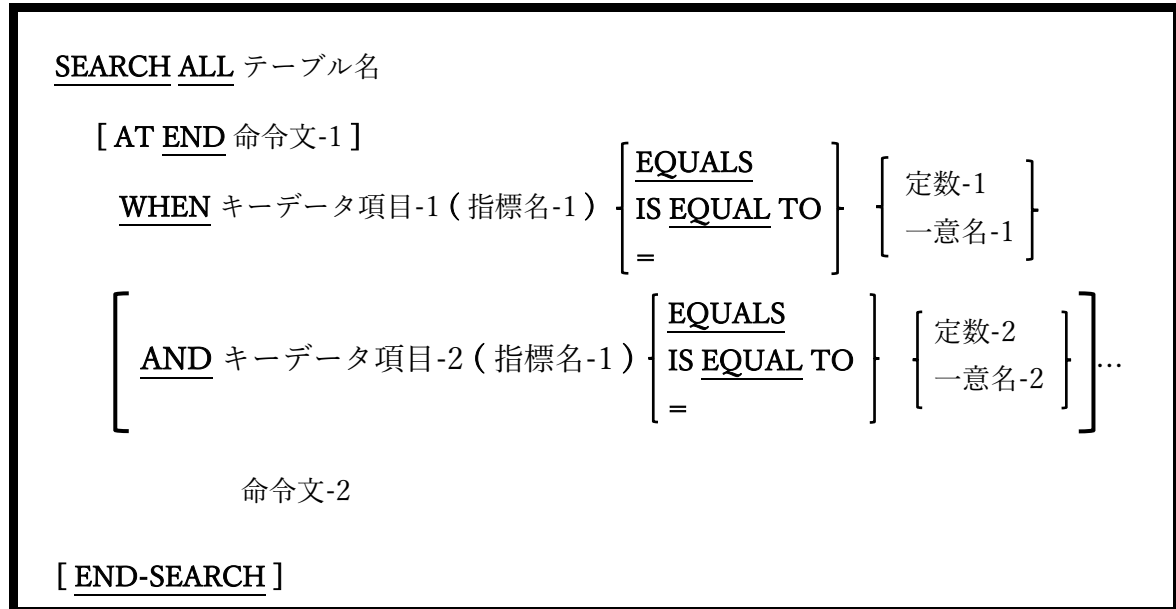
4. 探索プロセス中に条件式-1 が評価され、TRUE の場合は命令文-2 が実行された後に、制御は SEARCH 文の次に移る。
5. 複数の WHEN 句が存在する場合、それぞれの条件式-n が順番に評価され、最初に TRUE と評価された条件式に対応する命令文-n が実行された後に、制御は SEARCH

文の次に移る。

6. TRUE と評価される conditional-式-n が存在しない場合、指標名-1 の値は 1 ずつ増加する。指標名-1 の値がまだテーブル名の OCCURS 範囲内にある場合、WHEN 句が再度評価される。このプロセスは、WHEN 句の conditional-式-n が TRUE と評価されるまで、または指標名-1 の値がテーブル名の OCCURS 範囲内からなくなるまで継続する。
7. conditional-式-n が TRUE と評価されず、指標名-1 の値がテーブル名の OCCURS 範囲内にない場合、AT END 句の一部である命令文-1 が実行され、制御は SEARCH 文の次に移る。AT END 句がない場合、制御は単に SEARCH 文の次に移される。

## 6.38.2. SEARCH 文の書き方 2 — 二分探索(SEARCH ALL)

図 6-84-SEARCH 構文(二分探索(SEARCH ALL))



整列されたテーブルに対して二分探索を実行する。

1. テーブル名の定義には、OCCURS、ASCENDING(または DESCENDING)KEY、そして INDEXEDBY 句を含めなければならない。
2. SEARCH ALL 文を介してテーブルを探索できるようにするには、以下の項目が真である必要がある。
  - a. テーブルは上記 1 項の要件を満たしている。
  - b. テーブルに一つ以上の KEY 句がある時、テーブル内にその順序でデータが並んでいるわけではない。データの順序は KEY 句と一致している必要がある。<sup>22</sup>
  - c. テーブル内の二つのレコードが同じキー項目値を持つことはできない。また、テ

<sup>22</sup> もちろん、データの順序が KEY 句と一致しない場合は、テーブルソートを使って簡単に順序を揃えることができる(SORT 文の書き方 2-テーブルソートを参照)。

ーブルに複数の KEY 定義がある場合、テーブル内の二つのレコードが同じキー項目値の組み合わせを持つことはできない。

a に違反した場合、コンパイラは SEARCH ALL を拒否する。b または c、あるいはその両方に違反した場合、コンパイラによってメッセージは発行されないが、テーブルに対する SEARCH ALL の実行結果はおそらく正しくない。

3. キーデータ項目-1 およびキーデータ項目-2…(存在する場合)は、ASCENDING KEY 句または DESCENDING KEY 句を介して、テーブル名のキーとして定義する必要がある(上記 1 項を参照)。
4. 指標名-1 は、テーブル名の最初の INDEXED BY データ項目である。
5. SEARCH 文の書き方 1 とは異なり、WHEN 句は必須である。
6. 指定できる WHEN 句は一つのみである。AND 句の数に制限はないが、キー項目より WHEN 句および AND 句を多く指定することはできない。各 WHEN 句および AND 句は、異なるキー項目を参照する必要がある。
7. WHEN 句の機能は、AND 句とともに、最初の INDEXED BY 項目によって索引付けされたテーブルのキー項目を指定された定数または一意名の値と比較して、テーブルで目的の記述項を見つけることである。テーブルの索引は最小限のテストを必要とする方法で、SEARCH ALL 文によって自動的に変更される。
8. SEARCH ALL 文の内部処理は、初めに内部の「最初」および「最後」のポインタを、テーブルの最初と最後の記述項位置に設定し、次のように処理される。<sup>23</sup>
  - a. 「最初」と「最後」の中間の記述項が識別される。これを「現在の」記述項と呼

---

<sup>23</sup> これは、純粋な教育ツールとして意図されたアルゴリズムを簡略化した考え方であって、実装して機能させるためには、厄介ではあるが詳細を追加する必要がある(ルール「a」で「現在」のエントリが 12.5 であると識別されたときどうするか等)。

び、テーブル記述項の場所が指標名-1 に保存されるように設定する。

- b. WHEN 句(および AND 句)が評価される。目的の定数または一意名の値とキーを比較すると、次の三つのうちいずれかの結果になる。
  - i. キーと値が一致する場合、命令文 2 が実行された後、制御は SEARCH ALL の次の文に移る。
  - ii. キーが値よりも小さい場合、検索されるテーブル記述項は、テーブルの「現在」から「最後」の範囲内でのみ発生する可能性があるため、新しい「最初の」ポインタ値が設定される。(この場合「現在の」ポインタとして設定される)。
  - iii. キーが値よりも大きい場合、検索されるテーブル記述項は、テーブルの「最初」から「現在」の範囲内でのみ発生する可能性があるため、新しい「最後の」ポインタ値が設定される(この場合「現在の」ポインタとして設定される)。
- c. 新しい「最初」と「最後」のポインタが、古い「最初」と「最後」のポインタと異なる場合は、さらに検索する必要があるため、手順「a」に戻って検索を続ける。
- d. 新しい「最初」と「最後」のポインタが、古い「最初」と「最後」のポインタと同じである場合、テーブルは使い果たされているため検索されている記述項は見つからない。命令文 1 が実行された後、制御は SEARCH ALL の次の文に移る。

上記のアルゴリズムの効果は、特定の記述項が存在するかどうかを判断するために、テーブル内のごく一部の要素をテストする必要があることである。これは、SEARCH ALL が記述項をチェックするたび、テーブル内に残っている記述項の半分を破棄するために行われる。

コンピュータ研究者は、二つの探索方法を次のように比較する：

- 順次探索(書き方 1)では、記述項を見つけるために平均  $n / 2$  回、最悪の場合は  $n$  回の探索が必要であり、記述項が存在しないことを示す時も  $n$  回の探索が必要となる( $n$ =テーブル内の記述項の数)。
- 二分探索(書き方 2)では、記述項を見つけるために最悪の場合は  $\log_2 n$  回の探索、記述項が存在しないことを示す時でも  $\log_2 n$  回の探索が必要となる( $n$ =テーブル内の記述項の数)。

探索方法の違いについて、より具体的な考え方がある。テーブルに 1,000 個の記述項があるとする。順次探索(書き方 1)では、平均して 500 個をチェックして記述項を見つけるか、1,000 個全てを調べて記述項が存在しないことを確認する必要がある。二分探索では、記述項の数を 2 進数( $1,000_{10} = 1111101000_2$ )で表し、結果の桁数(10)を数える。これは、記述項を探索したり、記述項が存在しないことを確認したりするために必要な探索回数としては最小であり、かなりの改善されている。

## 6.39. SET

### 6.39.1. SET 文の書き方 1 — 環境設定

図 6-85-環境設定構文

|                                                                                                                                                                                                            |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\underline{\text{SET ENVIRONMENT}} \left[ \begin{array}{c} \text{整数-1} \\ \text{一意名-1} \end{array} \right] \underline{\text{TO}} \left[ \begin{array}{c} \text{整数-2} \\ \text{一意名-2} \end{array} \right]$ |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

プログラム内から環境値を簡単に設定することができる。

1. opensource COBOL プログラム内から生成または変更された環境変数は、そのプログラム(つまり CALL “SYSTEM”)によって生成されたすべてのサブシェルプロセスで使用できるが、opensource COBOL プログラムを開始したシェルまたはコンソールウィンドウには認識されない。
2. 環境変数を設定する手段としては、DISPLAY 文(6.14.3)を使うよりも、この方法は遥かに簡単で読みやすい。例えば、次の二つのコード順序は同じ結果を示す。

|                                                                                                                                 |                                           |
|---------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|
| <pre> DISPLAY "VALUE"   "VARNAME" UPON ENVIRONMENT-NAME END-DISPLAY DISPLAY   "VALUE" UPON ENVIRONMENT-VALUE END-DISPLAY </pre> | <pre> SET ENVIRONMENT "VARNAME" TO </pre> |
|---------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|

### 6.39.2. SET 文の書き方 2 — プログラムポインター設定

図 6-86-プログラムポインター設定構文

|                                                                                                                                                    |
|----------------------------------------------------------------------------------------------------------------------------------------------------|
| $\underline{\text{SET}} \text{ プログラムポインター-1 } \underline{\text{TO ENTRY}} \left[ \begin{array}{c} \text{定数-1} \\ \text{一意名-1} \end{array} \right]$ |
|----------------------------------------------------------------------------------------------------------------------------------------------------|

手続き部コードモジュールのアドレス、具体的には手続き部で宣言された記述項ポイントを取得できる。



1. 以前に他のバージョンの COBOL(特にメインフレームの実装)を使ったことがある場合は、サブルーチンの CALL が手続き部の段落または節の名前を引数として渡すのを見たことがあるかもしれないが、opensource COBOL では不可能である。その代わりに、この書き方の SET 文の使い方を知っておく必要がある。
2. program-pointer-1 はプログラムポインターとして使用しなければならない。
3. 定数-1 または一意名-1 の値には、プログラムの PROGRAM-ID、または ENTRY 文で指定された記述項ポイントを代入する必要がある。
4. この方法で手続き部コード領域のアドレスを取得すると、そのアドレスをサブルーチン(通常は C で書かれる)に渡して、必要な用途に使うことができる。動作中のプログラムポインターの例については、7.3.1.21 および 7.3.1.22 で説明する。

### 6.39.3. SET 文の書き方 3 – アドレス設定

図 6-87-アドレス設定構文

|                                                                                                                                                                                                                            |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\text{SET [ ADDRESS OF ] } \left\{ \begin{array}{l} \text{ポインター名-1} \\ \text{一意名-1} \end{array} \right\} \dots$ $\text{TO [ ADDRESS OF ] } \left\{ \begin{array}{l} \text{ポインター名-2} \\ \text{一意名-2} \end{array} \right\}$ |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

データ項目の内容ではなく、アドレスを処理するために使われる。

1. TO の前に ADDRESS OF 句がある場合、SET 文を使って連絡節または BASED データ項目のアドレスを変更する。この句がない場合は、一つ以上の USAGE POINTER データ項目にアドレスが割り当てられる。
2. TO の後に ADDRESS OF 句がある場合、一意名-1 に割り当てられるアドレス、またはポインター名-1 に格納されるアドレスとして、一意名-2 のアドレスを SET 文が識別する。この句がない場合は、ポインター名-2 の内容がアドレスに割り当てられる。

#### 6.39.4. SET 文の書き方 4 — インデックス設定

図 6-88-インデックス設定構文

|                                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------|
| $\underline{\text{SET}} \text{ 指標名-1 } \underline{\text{TO}} \left[ \begin{array}{l} \text{定数-1} \\ \text{一意名-1} \end{array} \right]$ |
|---------------------------------------------------------------------------------------------------------------------------------------|

USAGE INDEX データ項目に値を割り当てる。

1. 指標名-1 はインデックスである必要がある。または、指標名-1 はテーブル内で INDEXED BY 句と識別される必要がある。

#### 6.39.5. SET 文の書き方 5 — UP / DOWN 設定

図 6-89-UP/DOWN 設定構文

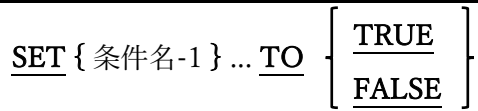
|                                                                                                                                                                                                 |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\underline{\text{SET}} \left[ \begin{array}{l} \text{指標名-1} \\ \text{ポインター-} \end{array} \right] \left[ \begin{array}{l} \underline{\text{UP}} \\ \underline{\text{DOWN}} \end{array} \right]$ |
| $\underline{\text{BY}} [\underline{\text{LENGTH OF}}] \left[ \begin{array}{l} \text{定数-1} \\ \text{一意名-2} \\ \text{関数 1} \end{array} \right]$                                                   |

インデックスまたはポインタの値を指定された値の分だけインクリメントまたはデクリメントするために使われる。

1. 指標名-1 はインデックスでなければならない。ポインター-1 はポインターまたはプログラムポインターである必要がある。
2. 指標名-1 が指定されている場合、一般的に UP または DOWN の値を 1 ずつ設定する。通常指標名-1 はテーブルの要素を順番にウォークスルーするために使われる。

### 6.39.6. SET 文の書き方 6 — 条件名設定

図 6-90-条件名設定構文



SET { 条件名-1 } ... TO  $\left[ \begin{array}{c} \underline{\text{TRUE}} \\ \underline{\text{FALSE}} \end{array} \right]$

レベル 88 条件名の TRUE / FALSE 値を指定することができる。

1. 指定された条件名を TRUE / FALSE 値に設定することで、実際には、条件名データ項目が従属する親データ項目に値を割り当てることになる。
2. TRUE を指定すると、各々の親データ項目に割り当てられる値は、条件名の定義で指定された最初の値になる。
3. SET 文で FALSE を指定すると、各々の親データ項目に割り当てられる値は、条件名の定義の FALSE 句によって指定された値になる。条件名-1 のオカレンスに FALSE 句がない場合、SET 文はコンパイラによって拒否される。

### 6.39.7. SET 文の書き方 7 – スイッチ設定

図 6-91-スイッチ設定構文

|                                                                                                                                                                           |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\underline{\text{SET}} \{ \text{ニ一モニツク名-1} \} \dots \underline{\text{TO}} \left\{ \begin{array}{c} \underline{\text{ON}} \\ \underline{\text{OFF}} \end{array} \right\}$ |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

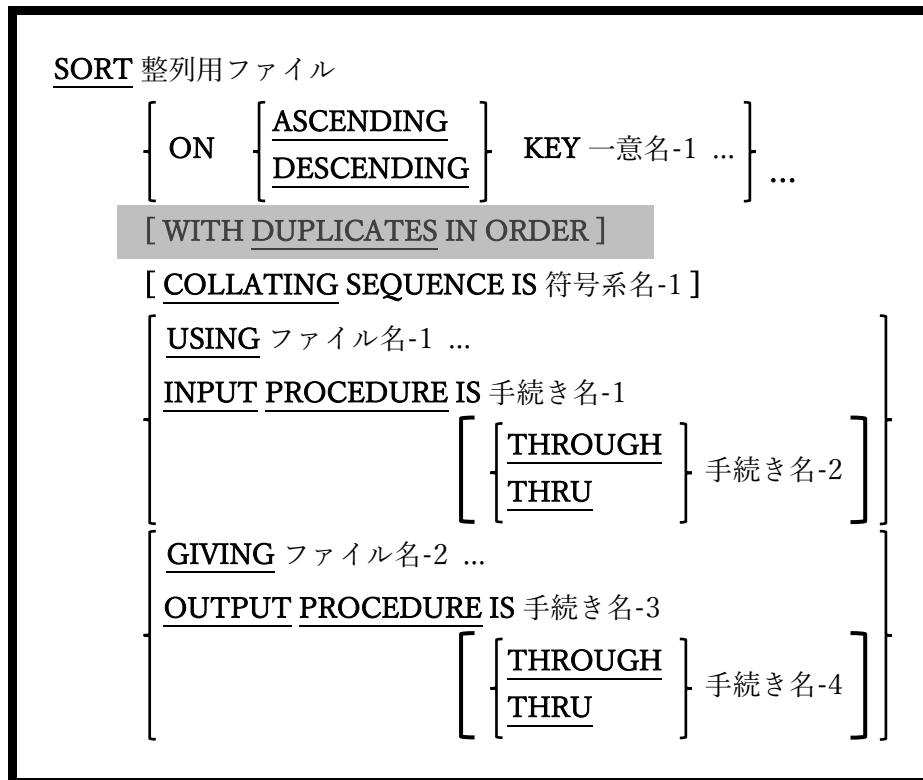
スイッチをオンまたはオフにする。

1. スイッチは、特殊名段落を使って定義される。詳細については、4.1.4 で説明している。

## 6.40. SORT

### 6.40.1. SORT 文の書き方 1 — ファイルソート

図 6-92-SORT 構文(ファイルソート)



一つ以上のキー項目に従って、大量のデータを整列することができる。

1. SORT 文で指定された整列ファイルは、データ部のファイル節でソート記述(SD)を使って定義する必要がある(5.2 を参照)。このファイルは「整列ファイル」と呼ばれる。
2. 指定する場合、ファイル名-1 およびファイル名-2 は、ORGANIZATION LINE SEQUENTIAL または ORGANIZATION RECORD BINARY SEQUENTIAL ファイルを参照する必要がある。これらのファイルは、データ部のファイル節のファイル記述(FD)を使って定義する必要がある(5.1 を参照)。ファイル名-1 とファイル名-2 に同じファイルを使うことができる。

3. 一意名-1…項目は、整列ファイルのレコード内の項目として定義する必要がある。
4. WITH DUPLICATES IN ORDER 句は互換性の目的でサポートされているが、機能はしない。
5. 整列ファイル(1 項を参照)が OPEN または CLOSE されることはない。
6. SORT 文は次の 3 段階の働きがある。

ステージ 1(入力フェーズ)：

- a. 整列されるデータは、整列ファイルにロードされる。USING 句で指定されたファイルの内容全体を取得するか、手続き名 1 または手続き名-1 THRU 手続き名-2 として定義された INPUT PROCEDURE を使うことによって達成される。
- b. USING を指定する場合、SORT の実行時にファイル名-1…を OPEN にすることはできない。
- c. INPUT PROCEDURE を使うと、整列されるレコードは必要なロジックを用いて生成され、RELEASE 文(6.34)を使うことで整列ファイルに一度につき一つずつ手動で書き込まれる。
- d. INPUT PROCEDURE 内で実行された STOP RUN、EXIT PROGRAM、または GOBACK は、現在実行中のプログラムと SORT 文を終了する。
- e. INPUT PROCEDURE から制御を移す GO TO 文は、SORT 文を終了するが、GO TO が制御を移した位置からプログラムの実行を継続できるようにする。GO TO を使って INPUT PROCEDURE を中止すると、再開することはできなくなるが、SORT 文自体を再実行することはできる。この方法で SORT 文を再起動すると、以前整列ファイルにリリースされたレコードはすべて失われてしまう。***GO TO を使って整列を早期に終了したり、以前に中止した SORT 文を再開したりす***

ることは、優れたプログラミングとは見なされないため、回避しなければならない。

- f. データが整列ファイルにロードされると、実際には動的に割り当てられたメモリにバッファリングされる。整列されるデータの量が使用可能なソートメモリ量 (128 MB)<sup>24</sup> を超える場合にのみ、実際のディスクファイルが割り当てられて使用される。これらの「整列作業ファイル」については、後ほど説明する。
- g. INPUT PROCEDURE は、手続き名-2(ない場合は手続き名-1)の最後の文を過ぎた後、制御のフォールスルーによって暗黙的に終了するか、手続き名-2(ない場合は手続き名-1)で実行される EXIT SECTION / EXIT PARAGRAPH を介して明示的に終了する。INPUT PROCEDURE が終了したところで、入力フェーズが完了する。
- h. INPUT PROCEDURE の範囲内では、ファイルの SORT、MERGE(6.27)、または RETURN(6.35)を実行できない。

ステージ 2(ソートフェーズ)：

- a. 整列は、(存在する場合は)SORT 文で指定された COLLATING SEQUENCE に従って、SORT 文内の ASCENDING KEY または DESCENDING KEY によって定義した順序でデータレコードを配置することで処理が行われる。何も定義されていない場合は、実行用計算機段落によって、PROGRAM COLLATING SEQUENCE が指定、または暗示される。キーは、レベル 78 またはレベル 88 のデータ項目を除いて、サポートされているものであれば、任意のデータ型と USAGE を設定することができる。
- b. 例えば、一連の金融取引の流れを整列してみると、SORT 文は次のようになる。

---

<sup>24</sup> 整列プロセスにはメモリを割り当てるためのランタイム環境変数(COB\_SORT\_MEMORY)がある(7.2.4 を参照)。

**SORT Sort-File****ASCENDING KEY Transaction-Date****ASCENDING KEY Account-Number****DESCENDING KEY Transaction-Amount**

·  
·  
·

この SORT 文の効果は、すべての取引を、取引が発生した日付の昇順(過去から最新へ)に整列することである。このプログラムを利用している企業が廃業しない限り、特定の日付で多くの取引が発生する可能性があるため、同じ日付の取引の各グループ内で、取引が行われた口座番号の昇順でサブソートされる。特定の日付に特定の口座で複数の取引が行われる可能性は非常に高いため、第 3 レベルのサブソートでは、同じ日付の同じ口座のすべての取引を、実際の取引額の降順(最高額から最低額へ)に整列する。2009 年 8 月 31 日に口座 # 12345 で 100.00 ドルの取引が二件以上記録された場合、整列キーに追加の「レベル」が指定されていないため、これらの取引が互いにどのように順序付けられているかを正確に予測する方法がない。

- c. opensource COBOL は、メインフレームコンピュータシステムのように、大容量で高性能な(そして高額な)整列用パッケージを使わないが、利用している SORT アルゴリズム <sup>25</sup>はこのタスクには十分すぎるほどである。

ステージ 3(出力フェーズ) :

- a. ソートフェーズが完了すると、GIVING 句が指定されている場合は整列済みデータがファイル名-2 に書き込まれるか、OUTPUT PROCEDURE を使って手続き名-3 または手続き名-3 THRU 手続き名-4 として定義される。
- b. GIVING 句を指定する場合、SORT 文の実行時にファイル名-2…を OPEN にして

---

<sup>25</sup> opensource COBOL ソートルーチンは、opensource COBOL ランタイムライブラリから完全に補うことができる。



はならない。

- c. OUTPUT PROCEDURE を使用する場合、整列済みレコードは、RETURN 文 (6.35)を使うことで整列ファイルに一度につき一つずつ手動で読み取られる。
  - d. OUTPUT PROCEDURE 内で実行された STOPRUN、EXIT PROGRAM、または GOBACK は、実行中のプログラムと SORT 文を終了する。
  - e. 制御を OUTPUT PROCEDURE から転送する GO TO 文は SORT 文を終了するが、GO TO が制御を転送した位置からプログラムの実行を継続できるようにする。GO TO を使って OUTPUT PROCEDURE を中止すると、再開することはできないが、SORT 文自体を再実行することはできる。この方法で SORT 文を再起動すると、整列ファイルから未返却のレコードはすべて失われてしまう。**GO TO を使って整列を早期に終了したり、以前に中止した SORT 文を再開したりすることは、優れたプログラミングとは見なされないため、回避しなければならない。**
  - f. OUTPUT PROCEDURE は、手続き名-4(ない場合は手続き名-3)の最後の文を過ぎた後、制御のフォールスルーによって暗黙的に終了するか、手続き名-4(ない場合は手続き名-3)で実行される EXIT SECTION / EXIT PARAGRAPH を介して明示的に終了する。OUTPUT PROCEDURE が終了したところで、出力フェーズおよび SORT 文自体が完了する。
  - g. OUTPUT PROCEDURE の範囲内では、ファイルの SORT、MERGE(6.27)、または RELEASE(6.34)を実行できない。
7. 整列されるデータの量によってディスク作業ファイルが必要な場合、TMPDIR、TMP、または TEMP 環境変数(7.2.4 を参照)によって定義されたフォルダー内のディスクに自動的に割り当てられる。ディスクファイルは、プログラムの実行終了時に自動的にパージされることはない。一時的な整列用ファイルは、自分で、または整列の終了時にプログラム内から削除する場合に備えて、「cobxxxx.tmp」という名前が付けられる。

## 6.40.2. SORT 文の書き方 2 – テーブルソート

図 6-93-SORT 構文(テーブルソート)

SORT テーブル名

{ ON { ASCENDING  
DESCENDING } KEY 一意名-1 ... } ...

[ WITH DUPLICATES IN ORDER ]

[ COLLATING SEQUENCE IS 符号系名-1 ]

一つ以上のキー項目に従って、比較的少量のデータ、つまり、データ部のテーブルに含まれるデータを整列する。

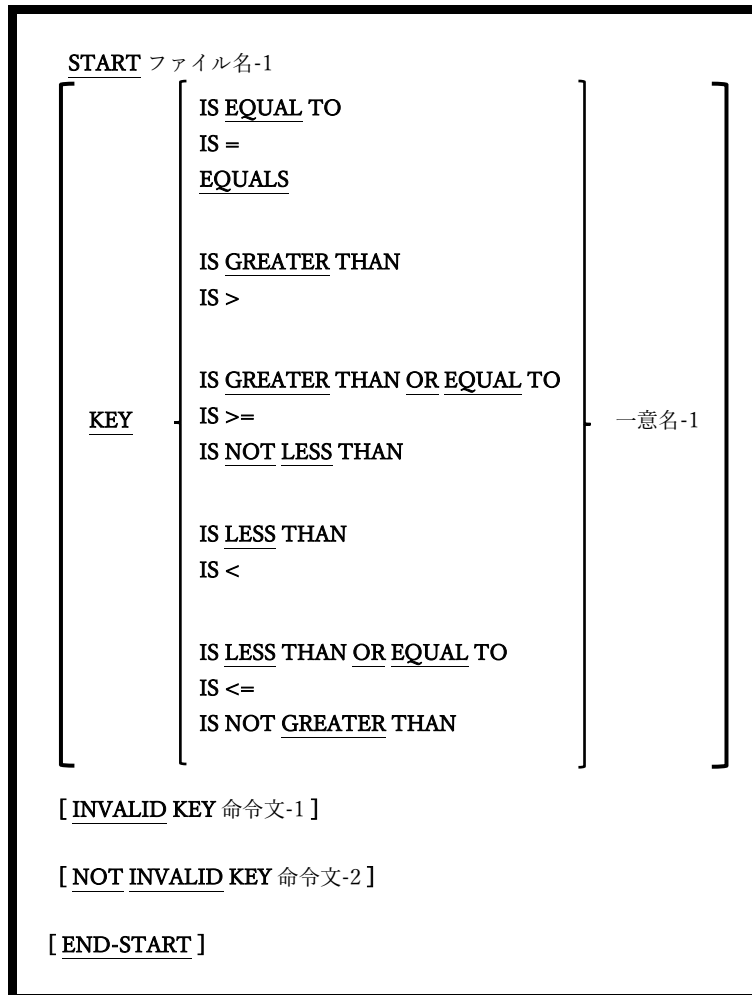
1. テーブル名データ項目には、OCCURS 句が必要である。
2. 一意名-1…項目が存在する場合は、テーブル名に従属するデータ項目として定義する必要がある。
3. WITH DUPLICATES IN ORDER 句は互換性の目的でサポートされているが、機能はしない。
4. テーブル名内のデータは、SORT 文で作成されたキー指定に従って所定の位置で整列される(つまり、整列ファイルは必要ない)。
5. 現在、SORT 文でキー指定が行われていないテーブルソートはサポートされておらず、コンパイラによって拒否される。
6. 整列は、(存在する場合は)SORT 文で指定された COLLATING SEQUENCE に従って、SORT 文内の ASCENDING KEY または DESCENDING KEY によって定義した順序でデータレコードを配置することで処理が行われる。何も定義されていない場合は、実行用計算機段落によって、PROGRAM COLLATING SEQUENCE が指定、または暗示される。キーは、レベル 78 またはレベル 88 のデータ項目を除いて、サポー

トされているものであれば、任意のデータ型と USAGE を設定することができる。

7. SORT 文はテーブル名内の所定の位置で実行されるため、整列ファイルは必要ない。

## 6.41. START

図 6-94-START 構文



START 文は、後続の順次読み取り操作のためのファイル内の論理開始点を定義する。

1. ファイル名-1 は、ORGANIZATION RELATIVE または ORGANIZATION INDEXED ファイルである必要がある。
2. ファイル名-1 は、ACCESS MODE DYNAMIC または ACCESS MODE SEQUENTIAL が SELECT で指定されている必要がある。
3. ファイル名-1 は START 文の実行時に、INPUT モードまたは I-O モードのいずれか

で OPEN(6.31)の状態である必要がある。

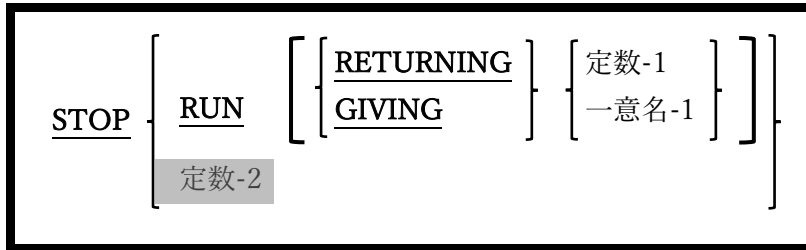
4. KEY 句が指定されていない場合、「**KEY IS EQUAL TO** 一意名-1」が指定されたとみなす。
5. ファイル名-1 が ORGANIZATION RELATIVE ファイルの場合、一意名-1 はファイルの RELATIVE KEY でなければならない(4.2.1.2 を参照)。
6. ファイル名-1 が ORGANIZATION INDEXED ファイルの場合、一意名-1 はファイルの RECORD KEY または ALTERNATE RECORD KEY 項目の一つでなければならない(4.2.1.3 を参照)。
7. START 文が正常に実行された後、ファイル名-1 データへの内部レコードポインターは、ファイル名-1 に対して実行された後続の順次 READ 文が読み取られるように配置される。
  - a. 指定された関係チェックが EQUALTO、GREATER THAN、GREATER THAN OR EQUAL TO(または構文上同じもの)である場合に KEY 句による指定を満たす最初のレコード。
  - b. KEY 句による指定を満たす最後のレコードは、指定された関係チェックが LESS THAN または LESS THAN OR EQUAL TO(または構文上同じもの)であるということである。
8. START 文は、後続の順次 READ 文のためにファイルを配置するだけであり、実際にファイル名-1 の 01 レベルのレコードに新しいデータを入力することはない。KEY 句を満たすレコードを読み取るには、START 文が成功した後に順次 READ 文を発行する必要がある。
9. START 文を実行中にエラーが発生した場合、ON INVALID KEY 句がトリガーされる(つまり命令文-1 が実行される)。このようなエラーは、入出力エラーまたは「キーが

存在しない」エラー(ファイルステータス 23)である可能性があり、KEY 句の要件を満たすレコードが存在しないことを示す。

10. START 文を実行中にエラーが発生しなかった場合、NOT INVALID KEY 句がトリガーされ、命令文-2 が実行される。
11. START 文が目的のレコードを見つけ(または見つけなくても)、指定された命令文-1 または命令文-2 を実行すると(または実行しなくても)、制御は START に続く次の文に移る。

## 6.42. STOP

図 6-95-STOP 構文



STOP 文はプログラムを停止し、オペレーティングシステムに制御を戻す。

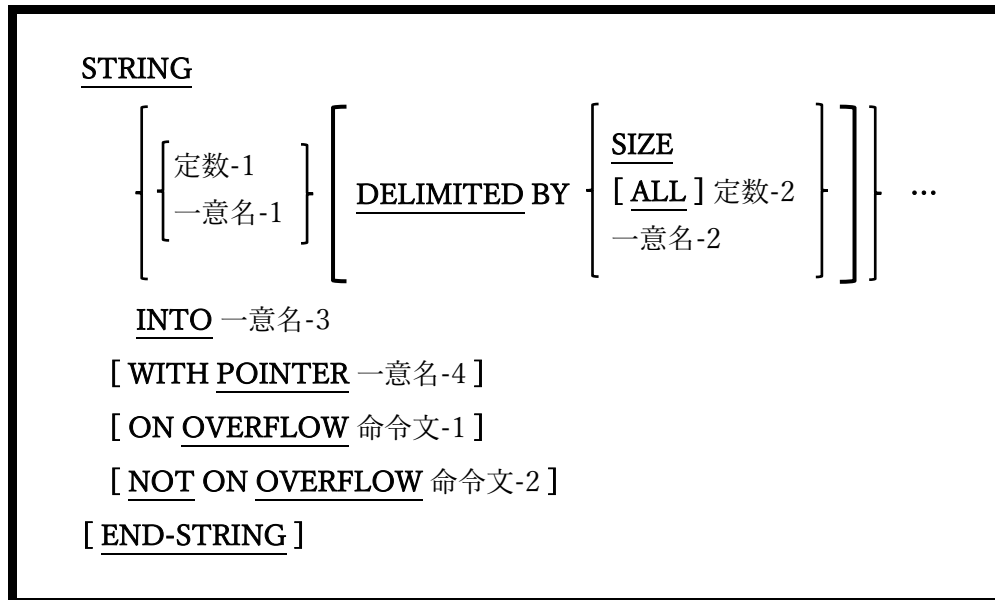
1. RETURNING 句と GIVING 句は同意義のものとして利用できる。
2. 定数-2 オプションは構文的にサポートされているが、廃止されているため、使用すると(警告とともに)拒否されてしまう。
3. RETURNING 句または GIVING 句を使うと、プログラムは数値リターンコードをオペレーティングシステムに返すことができ、リターンコードの値は、-2147483648 から+2147483647 の範囲にすることができる。
4. 以下の二つのコードは同じものである。リターンコードがオペレーティングシステムに返される、二つの異なる方法を以下に示す：

```
STOP RUN RETURNING 16
```

```
MOVE 16 TO RETURN-CODE
STOP RUN
```

## 6.43. STRING

図 6-96-STRING 構文



STRING 文は、複数の文字列のすべて、または一部を連結して新しい文字列を形成するために使われる。

1. 定数-1、定数-2、一意名-1、一意名-2、および一意名-3 は、英数字の USAGE DISPLAY データとして明示的または暗黙的に定義しなければならない。これらの一意名はいずれも集団項目である可能性がある。
2. 一意名-4 は、ゼロより大きい値を持ち、編集されていない基本整数値のデータ項目である必要がある。
3. 各定数-1 /一意名-1 は送信項目と呼ばれ、一意名-3 は受け取り項目と呼ばれる。
4. 各送信項目の内容は文字ごとに受け取り項目にコピーされる。最初の送信項目は、WITH POINTER 句で指定された文字位置から始まる受け取り項目へコピーされる(文字位置には 1 から順に番号が振られる)。WITH POINTER 句が指定されていない場合は、1 が割り当てられる。2 番目の送信項目は、最初の項目によって転送された最後の文字の次の文字位置から始まる受け取り項目へコピーされる。



5. 受け取り項目の最後の文字位置が入力されると、現在の送信項目にコピーすべきデータが残っているかどうか、または処理すべき送信項目が残っているかどうかに関係なく、STRING 処理は終了する。
6. 送信項目に DELIMITED BY SIZE オプションが指定されている場合、送信項目の全体がコピーされる。DELIMITED BY 句が指定されていない場合、DELIMITED BY SIZE が割り当てられる。
7. 送信項目に SIZE オプションのない DELIMITED BY 句がある場合、一意名-2 または **すべての** 定数-2 で指定された文字順序が送信項目で見つかり、送信項目のコピーが終了する。
8. 受け取り項目(一意名-3)は、STRING 文の開始時に(SPACES またはその他の値に)初期化されることも、コピーされる送信項目の文字総数が受け取り項目のサイズより少ない場合に SPACE で埋められることもない。必要に応じて、STRING を実行する前に受け取り項目を自分で明示的に INITIALIZE 文(6.24)を使って初期化することができる。
9. 一意名-4 の値が 1 未満の場合、またはすべての送信項目が完全に処理される前に受け取り項目の空白が不足している場合、オーバーフロー状態になる。このような場合に ON OVERFLOW 句が存在する時、命令文-1 が実行される。
10. オーバーフロー条件がなく、NOT ON OVERFLOW 句が存在する場合は、命令文-2 が実行される。
11. STRING 文が終了して命令文が実行されると、制御は STRING 文に続く次の文に移る。

## 6.44. SUBTRACT

### 6.44.1. SUBTRACT 文の書き方 1 — SUBTRACT FROM

図 6-97-SUBTRACT 構文

```
SUBTRACT [[LENGTH OF] [定数-1
一意名-1]] ...
FROM { 一意名-2 [ROUNDED] } ...
[ON SIZE ERROR 命令文-1]
[NOT ON SIZE ERROR 命令文-2]
[END-SUBTRACT]
```

FROM(一意名-1 または定数-1)の前にあるすべての引数の算術合計を生成し、その合計から TO(一意名-2)の後にリストされている各一意名を減算する。

1. 一意名-1 および一意名-2 は、編集不可の数値データ項目でなければならない。
2. 定数-1 は数字定数でなければならない。
3. ROUNDED、ON SIZE ERROR および NOT ON SIZE ERROR 句は、ADD 文 (6.5.1) の場合と同じように使われる。

### 6.44.2. SUBTRACT 文の書き方 2 — SUBTRACT GIVING

図 6-98-SUBTRACT GIVING 構文

```
SUBTRACT [LENGTH OF [定数-1
一意名-1]] ...
[FROM 一意名-2]
 GIVING { 一意名-3 [ROUNDED] }...
[ON SIZE ERROR 命令文-1]
[NOT ON SIZE ERROR 命令文-2]
[END-SUBTRACT]
```

FROM(一意名-1 または定数-1)の前にあるすべての引数の算術合計を生成し、その合計を一意名-2 の内容から減算し、GIVING(一意名-3)の後にリストされた一意名の内容をその結果に置き換える。

1. 一意名-1 および一意名-2 は、編集不可の数値データ項目でなければならない。
2. 一意名-3 は数値データ項目でなければならないが、編集可能な場合もある。
3. 定数-1 は数字定数でなければならない。
4. ROUNDED、ON SIZE ERROR および NOT ON SIZE ERROR 句は、ADD 文 (6.5.1) の場合と同じように使われる。

### 6.44.3. SUBTRACT 文の書き方 3 — SUBTRACT CORRESPONDING

図 6-99-SUBTRACT CORRESPONDING 構文

```
SUBTRACT CORRESPONDING 一意名-1 FROM 一意名-2 [ROUNDED]
 [ON SIZE ERROR 命令文-1]
 [NOT ON SIZE ERROR 命令文-2]
 [END-SUBTRACT]
```

二つの一意名に従属して見つかったデータ項目の一致と対応する、個々の SUBTRACT FROM 文と同等のコードを生成する。

1. 対応する一致を識別するためのルールは、6.28.2 — MOVE CORRESPONDING で説明している。
2. ROUNDED、ON SIZE ERROR および NOT ON SIZE ERROR 句は、ADD 文 (6.5.1) の場合と同じように使われる。

## 6.45. SUPPRESS

図 6-100-SUPPRESS 構文



**SUPPRESS PRINTING**

opensource COBOL コンパイラによって構文的に認識されるが、RWCS(COBOL Report Writer)は現在 opensource COBOL でサポートされていないため、SUPPRESS 文は機能しない。

## 6.46. TERMINATE

図 6-101-TERMINATE 構文

**TERMINATE** 一意名-1…

opensource COBOL コンパイラによって構文的に認識されるが、RWCS(COBOL Report Writer)は現在 opensource COBOL でサポートされていないため、TERMINATE 文は機能しない。

## 6.47. TRANSFORM

図 6-102- TRANSFORM 構文

|                                                                                                                                                                                                  |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <u>TRANSFORM</u> 一意名-1 <u>FROM</u> $\left[ \begin{array}{c} \text{定数-1} \\ \text{一意名-2} \end{array} \right] \text{ TO } \left[ \begin{array}{c} \text{定数-2} \\ \text{一意名-3} \end{array} \right]$ |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

TRANSFORM 文は、データ項目の一連の文字をスキャンして置換する。それは「TO」句の前後の引数によって定義される。

1. 「TO」句の前に指定された定数-1 または一意名-2 はターゲット文字列と呼ばれ、置き換える一意名-1 の文字を定義する。
2. 「TO」句の後に指定された定数-2 または一意名-3 は置換文字列と呼ばれ、定数-1 または一意名-2 で指定された文字と置き換える一意名-1 の文字を定義する。
3. TRANSFORM 文は 1985 年の COBOL 標準で廃止され、その機能は INSPECT 文、具体的には CONVERTING 句(6.26)に含まれている。
4. 一意名-1 の内容が一文字ずつスキャンされる。その文字がターゲット文字列に含まれている場合、置換文字列内の(相対位置に)対応する文字が一意名-1 の内容を置換する。
5. 置換文字列の長さがターゲット文字列の長さを超える場合、超過分は無視される。
6. ターゲット文字列の長さが置換文字列の長さを超える場合、長さの差を補うために置換文字列の右側に空白が埋め込まれていると見なされる。

図 6-103-機能的な TRANSFORM 文

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DEMOTRANSFORM.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 Sample-Item PIC X(20) VALUE 'THIS IS A TEST'.
PROCEDURE DIVISION.
000-Main.
 TRANSFORM Sample-Item
 FROM 'ABCDEFGHIJKLMNOPQRSTUVWXYZ '
 TO 'ZYXWVUTSRQPONMLKJIHGFEDCBA '
 DISPLAY
 Sample-Item
 END-DISPLAY
 STOP RUN
 .
```

出力結果

GSRH RH Z GVHG



## 6.48. UNLOCK

図 6-104 - UNLOCK 構文

|                                                   |
|---------------------------------------------------|
| <u>UNLOCK</u> ファイル名-1 {<br>RECORD<br>RECORDS<br>} |
|---------------------------------------------------|

この文は、まだ書き込まれていないファイル I / O バッファを指定されたファイル(存在する場合)に同期し、指定されたファイルに属するレコードに対して保持されているレコードロックを解放する。

1. ファイル名-1 が SORT ファイルの場合、アクションは実行されない。
2. すべての opensource COBOL 実装がロックをサポートしているわけではない。それらが構築されたオペレーティングシステムと、opensource COBOL が生成されたときに使用されたビルドオプションによって異なる。<sup>26</sup> これらの opensource COBOL 実装の一つを使用するプログラムが UNLOCK を発行すると、プログラムは無視されてコンパイルメッセージは発行されない。必要に応じて、バッファ同期は引き続き行われる。

---

<sup>26</sup> このマニュアルの著者は、例えば、MinGW ビルド/ランタイム環境を利用する Windows 用の opensource COBOL ビルドを使い、高度なファイル入出力に Berkeley データベースモジュールを利用する。opensource COBOL ビルドは LOCKing をサポートしていないが、UNIX ビルドは一般的にレコードロックをサポートしている。

## 6.49. UNSTRING

図 6-105-UNSTRING 構文

UNSTRING 一意名-1

DELIMITED BY  $\left\{ \begin{array}{l} [\underline{ALL}] \text{ 定数-1} \\ \text{一意名-2} \end{array} \right\} \left[ \underline{OR} \left\{ \begin{array}{l} [\underline{ALL}] \text{ 定数-2} \\ \text{一意名-3} \end{array} \right\} \right] \dots$

INTO 一意名-4 [ DELIMITER IN 一意名-5 ] [ COUNT IN 一意名-6 ]  
[ 一意名-7 [ DELIMITER IN 一意名-8 ] [ COUNT IN 一意名-9 ] ] ...

[ WITH POINTER 一意名-10 ]

[ TALLYING IN 一意名-11 ]

[ ON OVERFLOW 命令文-1 ]

[ NOT ON OVERFLOW 命令文-2 ]

[ END-UNSTRING ]

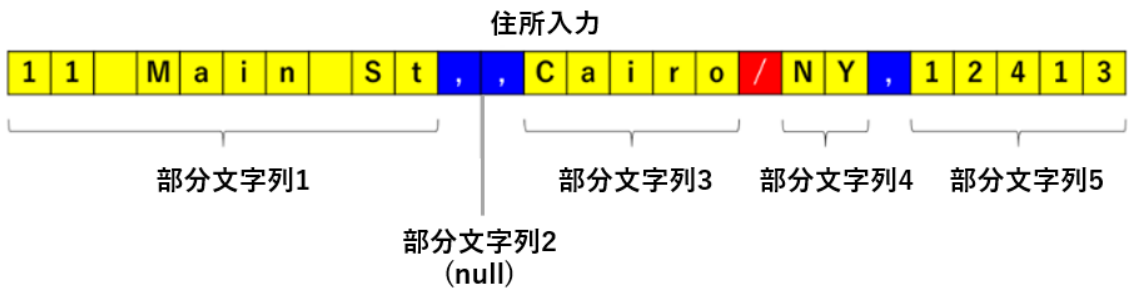
UNSTRING 文は文字列を解析し、そこから部分文字列を抽出する。

- 一意名-1 から一意名-5、一意名-7、および一意名-8 は、英数字の USAGE DISPLAY データとして明示的または暗黙的に定義する必要があり、これらの一意名はいずれも集団項目の可能性がある。
- 定数-1 および定数-2 は、英数字の定数でなければならない。
- 一意名-6 および一意名-9 から一意名-11 は、編集不可である基本の整数値項目でなければならない。
- 一意名-10 の値は 0 より大きい必要がある。
- 一意名-1 はソース文字列として知られ、一意名-4 と一意名-7 は宛先項目として知られている。

6. ソース文字列は、一意名-10 で示される文字位置から(WITH POINTER 句がない場合は 1 の場所から)始まる部分文字列に分割される。一意名-10 の初期値が 1 未満、またはソース文字列のサイズよりも大きい場合、オーバーフロー状態になる。オーバーフローについては、この後の 13 項で説明する。
7. 部分文字列は DELIMITED BY 句で指定された区切り文字列によって識別される。ALL オプションを使用すると、区切り文字順序を任意の長さの区切り文字定数のオカレンス順序にすることができるが、オプションがないと、各オカレンスは個別の区切り文字として扱われる。
8. 二つの連続する区切り文字順序は、空白の部分文字列を識別する。
9. ソース文字列が部分文字列に解析される例を次に示す：

```
UNSTRING Input-Address
 DELIMITED BY " " OR "/"
 INTO
 Street-Address DELIMITER D1 COUNT C1
 Apt-Number DELIMITER D2 COUNT C2
 City DELIMITER D3 COUNT C3
 State DELIMITER D4 COUNT C4
 Zip-Code DELIMITER D5 COUNT C5
END-UNSTRING
```

図 6-106- STRING 文の例



示されているサンプルデータから UNSTRING 文は合計 5 つの部分文字列を識別し、結果は次の MOVE 文が実行されたかようになる。

```

MOVE "11 Main St" TO Street-Address
MOVE "" TO Apt-Number27
MOVE "Cairo" TO City
MOVE "NY" TO State
MOVE "12413" TO Zip-Code

```

すべての宛先項目に入力するのに十分な部分文字列を識別できない場合、データが見つからない部分文字列は変更されない。

すべての部分文字列を受け取るのに十分な宛先項目が指定されていない場合、余分な部分文字列は「破棄」されるか「オーバーフロー」状態が存在する。オーバーフローについては、この後の 13 項で説明する。

10. 各宛先項目には、オプションの DELIMITER 句を使用することができる。DELIMITER 句が指定されている場合、一意名-5(または一意名-8)には、MOVE する宛先項目の部分文字列を識別するために使用される区切り文字列が含まれる。前に示した例を用いると、DELIMITER 一意名に対して次の暗黙の MOVE が発生する。

```

MOVE ", " TO D1
MOVE ",/" TO D2
MOVE "/" TO D3

```

<sup>27</sup> 空白文字列の MOVE は、空白の MOVE と同じである。

```

MOVE ",," TO D4
MOVE SPACES TO D528

```

11. 各宛先項目には、オプションの COUNT 句を使用することができる。COUNT 句が指定されている場合、一意名-6(または一意名-9)には、MOVE する宛先項目の部分文字列のサイズが含まれる。前に示した例を用いると、COUNT 一意名に対して次の暗黙の MOVE が発生する。

```

MOVE 10 TO C1
MOVE 0 TO C2
MOVE 5 TO C3
MOVE 2 TO C4
MOVE 5 TO C5

```

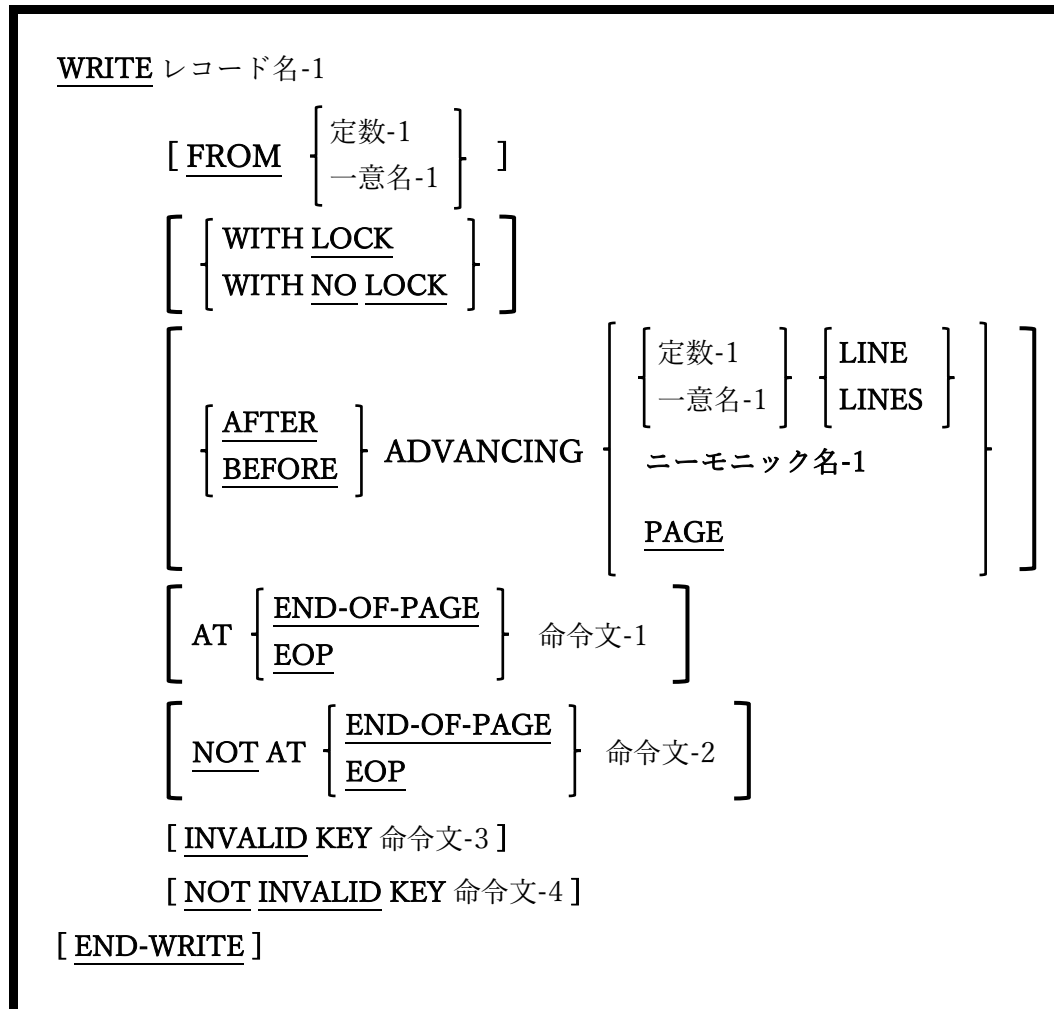
12. TALLYING 句(存在する場合)は、解析された部分文字列が宛先項目に MOVE されるたびに 1 ずつインクリメントされる。この項目をゼロに初期化する場合は、UNSTRING では行われなため、自分で行う必要がある。
13. オプションの ON OVERFLOW 句が存在する場合、オーバーフロー条件が発生すると(6 項および 7 項を参照)、命令文-1 が実行される。ON OVERFLOW 句がトリガーされた場合、NOT ON OVERFLOW 句(存在する場合)は無視される。
14. オプションの NOT ON OVERFLOW 句が存在せず、オーバーフロー条件が発生しない場合(6 項および 7 項を参照)、命令文-2 が実行される。NOT ON OVERFLOW 句がトリガーされた場合、ON OVERFLOW 句(存在する場合)は無視される
15. ソース文字列が解析されると、適切な宛先項目が更新され(DELIMITER / COUNT 項目とともに)、一意名-11(TALLYING)がインクリメントされ、ON OVERFLOW または NOT ON OVERFLOW 命令文が実行される。制御は UNSTRING 文に続く次の文に移る。

---

<sup>28</sup> 最後の部分文字列には常に空白の区切り文字があり、DELIMITER 項目に MOVE すると空白になる。

## 6.50. WRITE

図 6-107-WRITE 構文



WRITE 文は、OPEN ファイルに新しいレコードを書き込む。

- レコード名-1 は、OUTPUT、I-O または EXTEND に対して、現在も OPEN(6.31)状態であるファイルの、ファイル記述(FD-5.1 を参照)に従属する 01 レベルのレコードとして定義する必要がある。
- 定数-1 または一意名-1 は、英数字の USAGE DISPLAY データとして明示的または暗黙的に定義する必要がある。一意名-1 は集団項目の場合がある。

3. オプションの FROM 句を使用すると、レコード名-1 をファイルに書き込む前に、定数-1 または一意名-1 が暗黙的にレコード名-1 に MOVE する。
4. レコードの LOCK オプションについては 6.1.9.2 で説明している。
5. ADVANCING 句は、レポートが書き込まれる ORGANIZATION LINE SEQUENTIAL ファイルで使われることを目的としている。この句を他の ORGANIZATION で使用すると、コンパイラによって完全に拒否されるか (ORGANIZATION IS RELATIVE または ORGANIZATION IS INDEXED)、ファイルに不要な文字が書き込まれる可能性がある (ORGANIZATION IS RECORD BINARY SEQUENTIAL)。
6. ADVANCING n LINES 句は、書き込まれたレコードの前(AFTER ADVANCING)または書き込まれたレコードの後(BEFORE ADVANCING)のいずれかに、指定された数の改行(X "10")文字をファイルに導入する。
7. ORGANIZATION LINE SEQUENTIAL ファイルへの WRITE 文で ADVANCING 句が指定されていない場合、AFTER ADVANCING 1 LINE が指定されたとみなす。
8. ADVANCING PAGE 句は、書き込まれたレコードの前(AFTER ADVANCING)または書き込まれたレコードの後(BEFORE ADVANCING)のいずれかに、改ページ(X "0C")文字をファイルに導入する。
9. 書き込まれるファイルの FD に LINAGE 句(5.1)が含まれている場合、内部のラインカウンターはランタイムライブラリによって維持され、LINAGE 定義の LINES AT TOP および/または LINES AT BOTTOM 指定に対応するかたちで、適切な数の ASCII 改行文字がファイルに自動的に書き込まれる。
10. AT END-OF-PAGE 句と NOT AT END-OF-PAGE 句は、ファイル記述に LINAGE 句が含まれている ORGANIZATION LINE SEQUENTIAL または ORGANIZATION RECORD BINARY SEQUENTIAL ファイルに対してのみ有効である(5.1)。

11. WRITE 処理中にページ終了条件が発生した場合、AT END-OF-PAGE 句がトリガーされる(したがって命令文-1が実行される)。ページ終了条件は、WRITE文がデータ行または改行文字をファイルのページフッター領域内の行位置に導入したときに発生する(図 5-3 を参照)。
12. WRITE 処理中にページ終了条件が発生しなかった場合、NOT AT END-OF-PAGE 句がトリガーされる(したがって命令文-2 が実行される)。
13. 目的とする結果を得るには、ADVANCING 句と AT END-OF-PAGE 句の組合せの動作を理解する必要がある。そのために、これらの句を含む WRITE 文で発生する一連のイベントを次に示す：

- a. AFTER ADVANCING が指定されている場合：

AFTER ADVANCING PAGE が指定された場合、改ページ文字がファイルに書き込まれ、内部のページ終了スイッチが設定される。

それ以外の場合は、適切な数の改行文字(ADVANCING n LINES)がファイルに書き込まれる。内部の LINAGE カウンターが、改行によって論理ページの最大使用可能行数が使い果たされたことを示している場合、内部のページ終了スイッチが設定される。

- b. データレコードがファイルに書き込まれる。内部の LINAGE カウンターが、レコードの書き込みによって論理ページの最大使用可能行数が使い果たされたことを示している場合、内部のページ終了スイッチが設定される。

- c. BEFORE ADVANCING が指定されている場合：

BEFORE ADVANCING PAGE が指定された場合、改ページ文字がファイルに書き込まれ、内部のページ終了スイッチが設定される。



それ以外の場合は、適切な数の改行文字(ADVANCING n LINES)がファイルに書き込まれる。内部の LINAGE カウンターが、改行によって論理ページの最大使用可能行数が使い果たされたことを示している場合、内部のページ終了スイッチが設定される。

- d. 内部のページ終了スイッチが設定されていない場合、命令文-2(存在する場合)が実行される。

それ以外の場合(内部のページ終了スイッチが設定されている場合)、命令文-1(存在する場合)が実行される。

14. 上記 13 項を基に、AT END-OF-PAGE 句でページ見出しを自動生成できるサンプルコードは以下のようになる。

```
FD Report-File
 LINAGE IS 66 LINES
 WITH FOOTER AT 57
 LINES AT TOP 3
 LINES AT BOTTOM 3
.
.
.
OPEN OUTPUT Report-File
PERFORM Generate-Page-Header
.
.
.
WRITE Report-Rec AFTER ADVANCING 1 LINE
 AT END-OF-PAGE PERFORM Generate-Page-Header
END-WRITE
.
.
.
CLOSE Report-File
```

15. INVALIDKEY 句と NOT INVALID KEY 句は、ORGANIZATION RELATIVE または ORGANIZATION INDEXED ファイルで使われる WRITE 文でのみ有効である。
16. 書き込み中にエラーが発生した場合、ON INVALID KEY 句がトリガーされる(したがって命令文-3が実行される)。この場合、入出力エラーまたは「キーが既に存在している」エラー(ファイルステータス 22)である可能性があり、既に存在するレコードを書き込もうとしたことを示している。
17. 書き込み中にエラーが発生しなかった場合、NOT ON INVALID KEY 句がトリガーされる(したがって命令文 -4 が実行される)。

## 7. opensource COBOL システムインターフェース

### 7.1. opensource COBOL コンパイラの使い方(cobc)

#### 7.1.1. 解説

プログラムソースファイルの拡張子は「.cob」または「.cbl」が一般的である。

プログラムのファイル名は PROGRAM-ID の指定(大文字と小文字を含む)と完全に一致しなければならない。この理由については3章で説明している。

空白を PROGRAM-ID に含めることはできないため、プログラムのファイル名にも含めることはできない。

opensource COBOL コンパイラは、COBOL プログラムを C ソースコードに変換し、opensource COBOL のビルド時に指定された「C」コンパイラを使用してその C ソースコードを実行可能バイナリ形式にコンパイルし、その実行可能バイナリを、直接実行可能形式、静的リンク可能形式、または動的にロード可能な実行可能形式にリンクする。

opensource COBOL コンパイラの名称は「cobc」(Windows システムでは「cobc.exe」)である。

#### 7.1.2. 構文とオプション

次に、cobc コマンドの構文とオプションスイッチについて説明する。この情報は「cobc--help」のコマンドを入力することで表示することができる。

```
Usage: cobc [options] file...
```

```
Options:
```

```
--help Display this message
--version, -V Display compiler version
--info, -i Display compiler build information
-v Display the commands invoked by the compiler
-x Build an executable program
-m Build a dynamically loadable module (default)
-std=<dialect> Warnings/features for a specific dialect :
 cobol2002 Cobol 2002
 cobol85 Cobol 85
 ibm IBM Compatible
 mvs MVS Compatible
 bs2000 BS2000 Compatible
 mf Micro Focus Compatible
 default When not specified
See config/default.conf and config/*.conf
```

|                     |                                                                                  |
|---------------------|----------------------------------------------------------------------------------|
| -free               | Use free source format                                                           |
| -fixed              | Use fixed source format (default)                                                |
| -O, -O2, -Os        | Enable optimization                                                              |
| -g                  | Produce debugging information in the output                                      |
| -debug              | Enable all run-time error checking                                               |
| -o <file>           | Place the output into <file>                                                     |
| -b                  | Combine all input files into a single dynamically loadable module                |
| -E                  | Preprocess only; do not compile, assemble or link                                |
| -C                  | Translation only; convert COBOL to C                                             |
| -S                  | Compile only; output assembly file                                               |
| -c                  | Compile and assemble, but do not link                                            |
| -P                  | Generate preprocessed program listing (.lst)                                     |
| -Xref               | Generate cross reference through 'cobxref' (V. Coen's 'cobxref' must be in path) |
| -I <directory>      | Add <directory> to copy/include search path                                      |
| -L <directory>      | Add <directory> to library search path                                           |
| -l <lib>            | Link the library <lib>                                                           |
| -A <options>        | Add <options> to the C compile phase                                             |
| -Q <options>        | Add <options> to the C link phase                                                |
| -D <define>         | Pass <define> to the C compiler                                                  |
| -conf=<file>        | User defined dialect configuration - See -std=                                   |
| --list-reserved     | Display reserved words                                                           |
| --list-intrinsics   | Display intrinsic functions                                                      |
| --list-mnemonics    | Display mnemonic names                                                           |
| -save-temps(=<dir>) | Save intermediate files (default current directory)                              |
| -MT <target>        | Set target file used in dependency list                                          |
| -MF <file>          | Place dependency list into <file>                                                |
| -ext <extension>    | Add default file extension                                                       |
| -W                  | Enable ALL warnings                                                              |
| -Wall               | Enable all warnings except as noted below                                        |
| -Wobsolete          | Warn if obsolete features are used                                               |
| -Warchaic           | Warn if archaic features are used                                                |
| -Wredefinition      | Warn incompatible redefinition of data items                                     |
| -Wconstant          | Warn inconsistent constant                                                       |
| -Wparentheses       | Warn lack of parentheses around AND within OR                                    |
| -Wstrict-typing     | Warn type mismatch strictly                                                      |
| -Wimplicit-define   | Warn implicitly defined data items                                               |
| -Wcall-params       | Warn non 01/77 items for CALL params (NOT set with -Wall)                        |
| -Wcolumn-overflow   | Warn text after column 72, FIXED format (NOT set with -Wall)                     |
| -Wterminator        | Warn lack of scope terminator END-XXX (NOT set with -Wall)                       |
| -Wtruncate          | Warn possible field truncation (NOT set with -Wall)                              |
| -Wlinkage           | Warn dangling LINKAGE items (NOT set with -Wall)                                 |
| -Wunreachable       | Warn unreachable statements (NOT set with -Wall)                                 |
| -ftrace             | Generate trace code (Executed SECTION/PARAGRAPH)                                 |
| -ftraceall          | Generate trace code (Executed SECTION/PARAGRAPH/STATEMENTS)                      |
| -fsyntax-only       | Syntax error checking only; don't emit any output                                |
| -fdebugging-line    | Enable debugging lines ('D' in indicator column)                                 |
| -fsource-location   | Generate source location code (Turned on by -debug or -g)                        |
| -fimplicit-init     | Do automatic initialization of the Cobol runtime system                          |
| -fsign-ascii        | Numeric display sign ASCII (Default on ASCII machines)                           |
| -fsign-ebcdic       | Numeric display sign EBCDIC (Default on EBCDIC machines)                         |
| -fstack-check       | PERFORM stack checking (Turned on by -debug or -g)                               |
| -ffold-copy-lower   | Fold COPY subject to lower case (Default no transformation)                      |
| -ffold-copy-upper   | Fold COPY subject to upper case (Default no transformation)                      |
| -fwrite-after       | Use AFTER 1 for WRITE of LINE SEQUENTIAL (Default BEFORE 1)                      |
| -fnotrunc           | Do not truncate binary fields according to PICTURE                               |
| -ffunctions-all     | Allow use of intrinsic functions without FUNCTION keyword                        |
| -fmfcomment         | '*' or '/' in column 1 treated as comment (FIXED only)                           |
| -fnull-param        | Pass extra NULL terminating pointers on CALL statements                          |

2 章で説明したように、プログラムコンパイルユニットは、単一のソースファイルで順番に定義された複数のプログラムで構成されている場合がある。「cobc」コマンドで複数のソースファイルを指定することにより、「cobc」コマンドを 1 回実行するだけで複数のコンパイルユニットを処理することが可能になる。

### 7.1.3. 実行可能プログラムのコンパイル

最も簡単なコンパイルモードは、1 つ以上の opensource COBOL ソースファイルから単一の実行可能ファイルを生成することである。

```
cobc -x prog1.cbl prog2.cbl prog3.cbl
```

メインプログラムは、「prog1.cbl」ファイルにある最初のプログラムでなければならない。「prog1.cbl」の残りの部分、および「prog2.cbl」と「prog3.cbl」のすべては、サブプログラムまたはネストされたサブプログラムである必要がある。

これにより、必要なすべての COBOL プログラムが含まれている単一の実行可能ファイル (UNIX) または exe ファイル (Windows) が生成される。ただし、opensource COBOL、GMP、および BDB (または使用している opensource COBOL パッケージに組み込まれている他のファイル I/O モジュール) の動的ロード可能なランタイムライブラリは、実行時に引き続き使用可能である必要がある。

### 7.1.4. 動的にロード可能なサブプログラム

実行した時メモリに動的にロードされるサブプログラムは、次のように、cobc コマンドの「-m」オプションを使ってコンパイルする必要がある。

```
cobc -m sprog1.cbl
```

または

```
cobc -m sprog1.cbl sprog2.cbl sprog3.cbl
```

上記の最初のコマンドは動的にロード可能なモジュールを 1 つ生成し、2 番目の例は 3 つ生成する。

次のルールは、動的にロードされるモジュールとそれに含まれるサブルーチンに適用される。

1. 「xxxxxxx.cbl」または「xxxxxxx.cob」という名前のソースファイルから生成された動的にロード可能なモジュールは、UNIX システムでは「xxxxxxx.so」、Windows システムでは「xxxxxxx.dll」という名前になる。
2. 単一のサブプログラムのみを含む動的にロード可能なモジュールは、単一のプログラムのみを含む opensource COBOL ソースファイルから作成される。そのプログラムの PROGRAM-ID は、ソースコードのファイル名(マイナス「.cbl」または「.cob」)と動的にロード可能なモジュールのファイル名(拡張子「.so」または「.dll」を除く)と確実に一致する必要がある。
3. 複数のサブプログラムを含む動的にロード可能なモジュールは、複数のプログラムを含む単一の opensource COBOL ソースファイルから作成される。これらのプログラムの 1 つの PROGRAM-ID は、ソースコードのファイル名(マイナス「.cbl」または「.cob」)と動的にロード可能なモジュールのファイル名(マイナス「.so」または「.dll」)と確実に一致する必要がある。この PROGRAM-ID は、動的にロード可能なモジュールのプライマリ記述項ポイントである。
4. プログラムが動的にロード可能なモジュール内のサブプログラムを呼び出すとき
  - a. opensource COBOL ランタイムライブラリは、現在ロードされている動的にロード可能なすべてのモジュールで、サブプログラムの記述項ポイントを検索する(記述項ポイントは、CALL 文でコード化された定数または一意名(6.7 を参照))。その記述項ポイントは、動的にロード可能なモジュールを作成したソースファイル内の PROGRAM-ID(3 章)または記述項ポイント(6.16 章)のいずれかとして定義される。

- b. 記述項ポイントが見つかった場合、制御はそこに移され、サブプログラムが実行を開始する。
  - c. 記述項ポイントが見つからなかった場合、opensource COBOL ランタイムライブラリは「xxxxxxx.so」(UNIX)または「xxxxxxx.dll」(Windows)という名前のファイルを検索する。ここでの xxxxxxxx は目的のサブルーチン記述項ポイントを指す。
    - i. ファイルが見つかった場合は、ファイルがロードされ、そのファイル内の記述項ポイントに制御が移されるため、サブプログラムが実行を開始できる。
    - ii. ファイルが見つからなかった場合は、エラーメッセージ(「**libcob : モジュール'xxxxxxx'が見つかりません**」)が出力され、プログラムの実行が中止する。
5. 4 項は、複数の記述項ポイントを含む動的にロード可能なモジュールを使用したサブプログラミングに深い影響を及ぼす—モジュール内の他の記述項ポイントを呼び出す前に、モジュールの **プライマリ記述項ポイント** を正常に呼び出す必要がある(3 項を参照)。

「-x」オプションではなく「-m」オプション(上記コマンド参照)を使って、動的にロード可能なライブラリとしてメインプログラムを生成することも可能である。これらのメインプログラムを実行するには、7.2.2 で説明しているように、cobcrun コマンドを使う必要がある。

#### 7.1.5. 静的サブルーチン

opensource COBOL サブルーチンをアセンブラソースコードにコンパイルして、メインプログラムのコンパイル時に組み立てて繋げることもできる。このようなアセンブラソースファイルを作成するには、次のようにサブプログラムをコンパイルする。

```
cobc -S sprog1.cbl
```

(注: 「-S」は大文字で表記する)

これにより、「sprog1.s」というアセンブラソースファイルが作成される。複数の入力ファイルを指定すると、それぞれが独自の「.s」ファイルを作成する。メインプログラムをコンパイルするには、アセンブラソースファイルと組み合わせ、静的にリンクする。

```
cobc -x mainprog.cbl sprog1.s
```

複数のサブプログラムが必要な場合は、それらの「.s」ファイルをコマンドラインに追加するだけである。「.s」ファイルが指定されていないサブプログラムの記述項ポイントは、実行時に動的にロード可能なモジュールとして呼び出される。

#### 7.1.6. COBOL と C プログラムの結合

opensource COBOL と C 言語プログラム間のリンクは可能だが、プログラム間でデータを受け渡すためには、いずれかのプログラムで少し特別なコーディングが必要になる場合があり、次の 3 つが主な対処法である。問題について説明し、具体的にどのように対処するか、実際のプログラムコードを示す。

##### 7.1.6.1. opensource COBOL ランタイムライブラリの要件

COBOL 言語の他の実装と同様に、opensource COBOL はランタイムライブラリを使用する。特定の実行シーケンスで実行される最初のプログラム単位が opensource COBOL プログラムである場合、ランタイムライブラリの初期化は、C 言語プログラマにとって明確な方法である COBOL のコードによって実行される。ただし、C プログラム単位が最初に実行される場合は、opensource COBOL ランタイムライブラリの初期化を実行する負担が C プログラムにかかる。

##### 7.1.6.2. opensource COBOL と C の文字列割り当ての違い

どちらの言語も、文字列を固定長の連続した文字順序として格納する。



COBOL は、これらの文字順序を、データ項目の PICTURE 句によって課される特定の数量制限まで格納する。例：

```
01 LastName PIC X(15).
```

USAGE DISPLAY データ項目に含まれる文字列の長さは正確でなくてもよいが、PICTURE 句で許可されている文字数は常に正確である必要がある。上記の例では、「LastName」には常に正確に 15 文字が含まれる。もちろん、現在の LastName 値の一部として、0 から 15 までの末尾の空白が存在する可能性がある。

実際、C には「文字列」データ型がなく、配列の各要素が 1 文字である「char」データ型項目の配列として文字列を格納する。配列であるため、特定の「文字列」に格納できる文字数には上限がある。例：

```
char lastName[15]; /* 15 chars: lastName[0] thru lastName[14] */
```

C は、ある char 配列から別の char 配列に文字列をコピーしたり、特定の文字を文字列内で検索したり、ある char 配列を別の char 配列と比較したり、char 配列を連結したりするための、強力な文字列操作関数を提供する。これらの機能を可能にするために、文字列の論理的な終了を定義できる必要があった。C は、すべての文字列(char 配列)が NULL 文字 (x'00')で終了することを期待してこれを実現する。もちろん、プログラマはこれを強制されてはいないが、文字列を操作するために C 標準関数を使用するのであれば、実行したほうがよいだろう。

|                                    |                               |
|------------------------------------|-------------------------------|
| opensource COBOL Programmers Guide | opensource COBOL システムインターフェース |
|------------------------------------|-------------------------------|

### 7.1.6.3. C データ型と opensource COBOL USAGE 句の一致

これは非常に単純である。opensource COBOL と C のプログラマは、対応する C データ型と COBOL の USAGE 句を認識している必要がある。

表 7-1-C または opensource COBOL のデータ型の一致

| COBOL の USAGE 句<br>(PICTURE 句は使用できない)    | 占領する領域                | 保持できる数値                                                        | 対応するデータ型                                                                    |
|------------------------------------------|-----------------------|----------------------------------------------------------------|-----------------------------------------------------------------------------|
| BINARY-CHAR<br>BINARY-CHAR<br>UNSIGNED   | 1 バイト                 | 0~255                                                          | unsigned char                                                               |
| BINARY-CHAR SIGNED                       | 1 バイト                 | -128~+127                                                      | signed char                                                                 |
| BINARY-SHORT<br>BINARY-SHORT<br>UNSIGNED | 2 バイト                 | 0~65535                                                        | unsigned<br>unsigned int<br>unsigned short<br>unsigned short int            |
| BINARY-SHORT<br>SIGNED                   | 2 バイト                 | -32768~+32767                                                  | int<br>short<br>short int<br>signed int<br>signed short<br>signed short int |
| BINARY-LONG<br>BINARY-LONG<br>UNSIGNED   | 4 バイト                 | 0~4294967295                                                   | unsigned long<br>unsigned long int                                          |
| BINARY-LONG<br>SIGNED                    | 4 バイト                 | -2147483648<br>~<br>+2147483647                                | long<br>long int<br>signed long<br>signed long int                          |
| BINARY-C-LONG<br>SIGNED                  | 4 バイト<br>または<br>8 バイト | -2147483648<br>~<br>+2147483647<br>または<br>-9223372036854775808 | long<br>(USAGE BINARY-C-LONG の表 5-10 を参照)                                   |

|                                    |                               |
|------------------------------------|-------------------------------|
| opensource COBOL Programmers Guide | opensource COBOL システムインターフェース |
|------------------------------------|-------------------------------|

|                                            |        |                                                                         |                                                 |
|--------------------------------------------|--------|-------------------------------------------------------------------------|-------------------------------------------------|
|                                            |        | ~<br>+9223372036854775807                                               |                                                 |
| BINARY-DOUBLE<br>BINARY-DOUBLE<br>UNSIGNED | 8 バイト  | 0~18446744073709551615                                                  | unsigned long long<br>unsigned long long<br>int |
| BINARY-DOUBLE<br>SIGNED                    | 8 バイト  | -9223372036854775808<br>~<br>+9223372036854775807                       | long long int<br>signed long long int           |
| COMPUTATIONAL-1                            | 4 バイト  | $-3.4 \times 10^{38} \sim +3.4 \times 10^{38}$<br>(小数点以下 6 桁の精度)        | float                                           |
| COMPUTATIONAL-2                            | 8 バイト  | $-1.7 \times 10^{308} \sim +1.7 \times 10^{308}$<br>(小数点以下 15 桁の精度)     | double                                          |
| N/A(opensource COBOL<br>に相当するものなし)         | 12 バイト | $-1.19 \times 10^{4932} \sim +1.19 \times 10^{4932}$<br>(小数点以下 18 桁の精度) | long double                                     |

同じストレージサイズと値の範囲の組み合わせを定義できる、他の opensource COBOL の PICTURE 句または USAGE 句の組み合わせがある。しかし (COMP-1 と COMP-2 を除いて)、これらは C プログラムのデータ互換性のための ANSI2002 標準仕様であり、データが C プログラムと共有されている場合、opensource COBOL プログラマはこれを使用することに慣れておく必要がある (優れたドキュメントでもあり、データが C プログラムと「共有」されるという事実を強調している)。

様々な SIGNED 整数の USAGE 句で示されている最小値は、負の符号付きバイナリ値に 2 の補数表現を使用するコンピュータシステム (Windows PC でよく見られる CPU など) に適している。負の符号付きバイナリ値に 1 の補数表現を使用するコンピュータシステムでは、最小値が 1 大きくなる (例えば、-128 ではなく -127)。

#### 7.1.6.4. opensource COBOL メインプログラムの C サブプログラム呼び出し

C サブプログラムを CALL する opensource COBOL プログラムの例を次に示す。

図 7-2-opensource COBOL の C 呼び出し

| (maincob.cbl)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | (subc.c)                                                                                                                                                                                                                                                                                                                                                          |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| opensource COBOL メインプログラム                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | 呼び出される C サブプログラム                                                                                                                                                                                                                                                                                                                                                  |
| <pre> IDENTIFICATION DIVISION. PROGRAM-ID. maincob. DATA DIVISION. WORKING-STORAGE SECTION. 01 Arg1          PIC X(7). 01 Arg2          PIC X(7). 01 Arg3          USAGE BINARY-LONG. PROCEDURE DIVISION. 000-Main.     DISPLAY 'Starting cobmain'.     MOVE 123456789 TO Arg3.     STRING 'Arg1'            X'00'            DELIMITED SIZE            INTO Arg1     END-STRING.     STRING 'Arg2'            X'00'            DELIMITED SIZE            INTO Arg2     END-STRING.     CALL 'subc' 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. </pre> | <pre> #include &lt;stdio.h&gt;  int subc(char *arg1,          char *arg2,          unsigned long *arg3) {     char nul[7]="New1";     char nu2[7]="New2";     printf("Starting subc\n");     printf("Arg1=%s\n",arg1);     printf("Arg2=%s\n",arg2);     printf("Arg3=%d\n",*arg3);     arg1[0]='X';     arg2[0]='Y';     *arg3=987654321;     return 2; } </pre> |

考え方としては、2 つの文字列と 1 つのフルワードの符号なし引数をサブプログラムに渡し、サブプログラムにそれらを出力させ、3 つすべてを変更して、リターンコード 2 を呼び出し元に渡すことである。次に、呼び出し元は 3 つの引数を再表示し(2 つの BY REFERENCE 引数の変更のみ表示する)、リターンコードを表示して停止する。これら 2 つのプログラムは単純だが、必要な手法がよく説明されている。

COBOL プログラムが、null の文字列終了符が両方の文字列引数に存在することの確認方法に注意すること。

C プログラムは 3 つの引数に変更を加えようとしているため、関数の先頭で 3 つをポイン

ターとして宣言し、関数の本体で 3 番目の引数をポインターとして参照する。<sup>29</sup>

これらのプログラムは、次のようにコンパイルおよび実行される。以下の例では、ネイティブ C コンパイラを使用する opensource COBOL ビルドを備えた UNIX システムを想定している。この手法は、使用している C コンパイラやオペレーティングシステムに関係なく、同じように機能する。

```
$ cc -c subc.c
$ cobc -x maincob.cbl subc.o
$ maincob
Starting cobmain
Starting subc
Arg1=Arg1
Arg2=Arg2
Arg3=123456789
Back
Arg1=Arg1
Arg2=Yrg2
Arg3=+0987654321
Returned value=+000000002
$
```

null 文字は、実際は opensource COBOL の「Arg1」および「Arg2」データ項目にあることに注意すること。出力には表示されないが存在する。文字列を C プログラムに渡す場合、文字列項目の null 終了コピーを作成して C プログラムに渡すことを推奨する。

6.7 で説明したように、サブプログラムが opensource COBOL 以外の言語で記述されている場合、opensource COBOL のサブプログラム呼び出しでは、BY CONTENT 句を指定して、サブプログラムが引数を変更できないようにする必要がある。CALL する側のプログラムと CALL される側のプログラムの両方が opensource COBOL である場合、BY VALUE 句は BY CONTENT 句のより高速な代替手段になる。

---

<sup>29</sup> 実際には、2 つの文字列(char 配列)引数は選択できなかった。ポインターを表す「\*」を先頭に付けずに関数コードで参照していても、関数内でポインターとして定義する必要がある。

### 7.1.6.5. C メインプログラムの opensource COBOL サブプログラム呼び出し

ここでは前の章の 2 つの言語の役割が反転し、C メインプログラムが opensource COBOL サブプログラムを実行する。

図 7-3-C の opensource COBOL 呼び出し

| (mainc.c)<br>C メインプログラム                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | (subcob.cbl)<br>呼び出される opensource COBOL サブプログラム                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>#include &lt;libcob.h&gt; #include &lt;stdio.h&gt;  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_init(0,NULL) if cmdline args not     going     to COBOL */     returnCode = subcob(arg1,arg2,&amp;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; }</pre> | <pre>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 cobsb.cbl'.     DISPLAY 'Arg1=' Arg1.     DISPLAY 'Arg2=' Arg2.     DISPLAY 'Arg3=' Arg3.     MOVE 'X' TO Arg1 (1:1).     MOVE 'X' TO Arg2 (1:1).     MOVE 987654321 TO Arg3.     MOVE 2 TO RETURN-CODE.     GOBACK.</pre> |

C プログラムは opensource COBOL サブルーチンの前に最初に実行されるため、opensource COBOL ランタイム環境を初期化する負担はその C プログラムにあり、「libcob」ライブラリの一部である「cob\_init」関数を呼び出す必要がある。

「cob\_init」ルーチンへの引数は、プログラムの実行開始時にメイン関数に渡された引数の数と値のパラメータである。これらを opensource COBOL サブプログラムに渡すことにより、その opensource COBOL プログラムが、コマンドラインまたは個々のコマンドライン引数を取得できるようになる。それが必要なければ、「cob\_init(0,NULL);」を代わりに指定できる。

C プログラムは、「arg3」がサブプログラムによって変更されることを許可しているため、「&」を前に付けて BY REFERENCE 句による引数呼び出しを強制する。「arg1」と「arg2」は文字列(char 配列)であるため、自動的に参照渡しされる。

コンパイルプロセスとプログラム実行の出力を次に示す。以下の例では、GNU C コンパイ

ラを使用する opensource COBOL ビルドを備えた Windows システムを想定している。この手法は、使用している C コンパイラやオペレーティングシステムに関係なく、同じように機能する。

```
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=Xrg2
Arg3=987654321
Returned value=2
C:\Users\Gary\Documents\Programs>
```

第 1 引数が BY VALUE 句であることを opensource COBOL で記述したにも関わらず、BY REFERENCE 句であるかのように扱われたことに注意すること。C 呼び出し元から opensource COBOL サブプログラムに渡される文字列(char 配列)引数は、サブプログラムによって変更可能である。サブプログラムによって変更されないようにする場合は、データのコピーを渡すのが最善である。

ただし、3 番目の引数は異なる。これは配列ではないため、BY REFERENCE 句<sup>30</sup> または BY VALUE 句<sup>31</sup> のいずれかで渡すことができる。

---

<sup>30</sup> C 呼び出しプログラムでは、引数に「&」を使用する。COBOL サブプログラムで引数を BY REFERENCE 句として指定する。

<sup>31</sup> C 呼び出しプログラムでは、引数に「&」を使用してはいけない。COBOL サブプログラムで引数を BY VALUE 句として指定する。

|                                    |                               |
|------------------------------------|-------------------------------|
| opensource COBOL Programmers Guide | opensource COBOL システムインターフェース |
|------------------------------------|-------------------------------|

### 7.1.7. 重要な環境変数

次の表は、opensource COBOL プログラムのコンパイルで利用できる様々な環境変数を示している。

表 7-4-環境変数コンパイラ

| 環境変数                     | 使い方                                                                                                                                                     |
|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| COB_CC                   | opensource COBOL で使用する C コンパイラの名前に設定する。<br><br>この機能の利用は自己責任である—opensource COBOL ビルドが生成された C コンパイラを常に使用する必要がある。                                          |
| COB_CFLAGS <sup>32</sup> | cobc コンパイラから C コンパイラに渡すスイッチに設定する(cobc が指定するスイッチに加えて)。既定値は「 <b>-Iprefix/include</b> 」で、「prefix」は使用している opensource COBOL のインストールパスである。                    |
| COB_CONFIG_DIR           | opensource COBOL の「構成」ファイルが保存されているフォルダへのパスに設定する。構成ファイルの使用方法については、7.1.9 で説明する。                                                                           |
| COB_COPY_DIR             | プログラムに必要な COPY モジュールがプログラムと同じディレクトリに保管されていない場合は、この環境変数を COPY モジュールが含まれているフォルダに設定する (IBM メインフレームプログラマはこれを「SYSLIB」と認識する)。COPY モジュールの使用に関する追加情報については、7.1.8 |

<sup>32</sup> これらのスイッチは、高度なユーザによる特殊な状況での使用のみを目的としているため、使用は推奨していない。opensource COBOL の今後のリリースでは、cobc コマンドから C コンパイラやローダーに切り替えるためのより良い方法が導入される予定である。



|                                    |                               |
|------------------------------------|-------------------------------|
| opensource COBOL Programmers Guide | opensource COBOL システムインターフェース |
|------------------------------------|-------------------------------|

|                            |                                                                                                                                                           |
|----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
|                            | で説明する。                                                                                                                                                    |
| COB_LDADD                  | プログラムとリンクする必要がある標準ライブラリがつけられる場所を指定できる追加のリンカースイッチ(ld)に設定する。既定値は""(null)。                                                                                   |
| COB_LDFLAGS                | cobc コンパイラから C コンパイラに渡すリンカ/ローダ(ld)スイッチに設定する(cobc が指定するスイッチに加えて)。既定値は未設定。                                                                                  |
| COB_LIBS                   | プログラムとリンクする必要がある標準ライブラリがつけられる場所を指定するリンカースイッチ(ld)に設定する。既定値は「 <b>-Lprefix/lib-lcob</b> 」で、「prefix」は、使用している opensource COBOL バイナリが作成されたときに指定されたパスプレフィックスである。 |
| COBCPY                     | この環境変数は、コンパイラが COPY モジュールがつけられる場所を指定する追加手段を提供する(上記の COB_COPY_DIR も参照)。COPY モジュールの使用に関する追加情報については、7.1.8 で説明する。                                             |
| LD_LIBRARY_PATH            | 静的にリンクされたサブルーチンライブラリの使用を計画している場合は、この変数を、ライブラリを含むディレクトリへのパスに設定する。                                                                                          |
| TMPDIR<br>TMP<br>(この順番で確認) | 一時ファイルを作成するのに適したディレクトリ/フォルダに設定する。cobc によって作成された中間作業ファイルがここに生成される(不要になると削除される)。<br><br>通常 Windows システムでは、ログオン時に TMP 環境変数が設定される。 <u>別の</u> 一時フォ             |

|                                    |                               |
|------------------------------------|-------------------------------|
| opensource COBOL Programmers Guide | opensource COBOL システムインターフェース |
|------------------------------------|-------------------------------|

|  |                                                                            |
|--|----------------------------------------------------------------------------|
|  | <p>ルダを使用する場合は、TMPDIR を自分で設定すれことで、TMP に依存する他の Windows ソフトウェアを中断する心配はない。</p> |
|--|----------------------------------------------------------------------------|

### 7.1.8. コンパイル時のコピーブックの検索

opensource COBOL コンパイラは、以下のフォルダでコピーブック(COPY 文を介してコンパイルプロセスに持ち込まれたソースコードモジュール)を検索する。検索は以下の順序で実行され、コピーブックが見つかりと終了する。

- コンパイルされるプログラムが存在するフォルダ。
- 「-I」コンパイラスイッチ(7.1.2 を参照)で指定されたフォルダ。
- COBCPY 環境変数(7.1.7 を参照)で指定された各フォルダ。システムに適した区切り文字で区切ることによって、単一のフォルダあるいは複数のフォルダを指定することができる。<sup>33</sup> 複数のフォルダを指定した場合、環境変数で指定された順序で検索される。
- COB\_COPY\_DIR 環境変数(7.1.7 を参照)で指定されたフォルダ。

上記の各フォルダでコピーブック—例えば「COPY XXXXXXXX」—が検索されると、opensource COBOL コンパイラは次のいずれかの名前で順にコピーブックファイルを検索する。

- XXXXXXXX.CPY
- XXXXXXXX.CBL
- XXXXXXXX.COB
- XXXXXXXX.cpy

---

<sup>33</sup> opensource COBOL コンパイラがネイティブ Windows 環境用に構築されている場合は、セミコロン(:)を使用する。ただし、opensource COBOL コンパイラが Unix または Linux 環境用、または Cygwin や MinGW Unix「エミュレータ」を使った Windows 環境用に構築されている場合は、区切り文字としてコロンの文字(:)を使用する。

- XXXXXXXX.cbl
- XXXXXXXX.cob
- XXXXXXXX

UNIX システムでは COPY コマンドの大文字と小文字が区別される。「COPY copybookname」と「COPY COPYBOOKNAME」はどちらも、UNIX システムで「CopyBookName」コピーブックを見つけることはできない。opensource COBOL の Windows 実装では、Windows のバージョンと opensource COBOL ビルドオプションに応じて、コピーブック名の大文字と小文字が区別される場合とされない場合があるが、すべての環境で COPY コマンドを大文字と小文字を区別するものとして扱うのが最も安全である。

#### 7.1.9. コンパイラ構成ファイルの使い方

opensource COBOL は、コンパイラ構成ファイルを使って、コンパイルプロセスを制御する様々なオプションを定義する。これらの構成ファイルは、「-conf」コンパイルスイッチで指定されるか、COB\_CONFIG\_PATH 環境変数で定義されたフォルダにある。

以下は、「初期値」構成ファイル(「-conf」スイッチを指定しない場合に使用される)の逐語的なリストで、設定を表示する。

```
COBOL compiler configuration -*- sh -*-

Value: any string
name: "opensource COBOL"

Value: int
tab-width: 8
text-column: 72

Value: 'cobol2002', 'mf', 'ibm'
#
assign-clause: mf

If yes, file names are resolved at run time using environment variables.
For example, given ASSIGN TO "DATAFILE", the actual 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.
#
Value: 'yes', 'no'
filename-mapping: yes

Value: 'yes', 'no'
pretty-display: yes

Value: 'yes', 'no'
auto-initialize: yes

Value: 'yes', 'no'
complex-odo: no

Value: 'yes', 'no'
indirect-redefines: no

Binary byte size - defines the allocated bytes according to PIC
Value: signed unsigned bytes
----- -
'2-4-8' 1 - 4 2
5 - 9 4
10 - 18 8
#
'1-2-4-8' 1 - 2 1
3 - 4 2
5 - 9 4
10 - 18 8
#
'1--8' 1 - 2 1
3 - 4 2
5 - 6 3
7 - 9 4
10 - 11 5
12 - 14 6
15 - 16 7
17 - 18 8
binary-size: 1-2-4-8

Value: 'yes', 'no'
binary-truncate: yes

Value: 'native', 'big-endian'
binary-byteorder: big-endian

Value: 'yes', 'no'
larger-redefines-ok: no

Value: 'yes', 'no'
relaxed-syntax-check: no

Perform type OSVS - If yes, the exit point of any currently executing perform
is recognized if reached.

```

```

Value: 'yes', 'no'
perform-osvs: no

If yes, linkage-section items remain allocated
between invocations.
Value: 'yes', 'no'
sticky-linkage: no

If yes, allow non-matching level numbers
Value: 'yes', 'no'
relax-level-hierarchy: no

not-reserved:
Value: Word to be taken out of the reserved words list
(case independent)

Dialect features
Value: 'ok', 'archaic', 'obsolete', 'skip', 'ignore', 'unconformable'
author-paragraph: obsolete
memory-size-clause: obsolete
multiple-file-tape-clause: obsolete
label-records-clause: obsolete
value-of-clause: obsolete
data-records-clause: obsolete
top-level-occurs-clause: skip
synchronized-clause: ok
goto-statement-without-name: obsolete
stop-literal-statement: obsolete
debugging-line: obsolete
padding-character-clause: obsolete
next-sentence-phrase: archaic
eject-statement: skip
entry-statement: obsolete
move-noninteger-to-alphanumeric: error
odo-without-to: ok

```

## 7.2. opensource COBOL プログラムの実行

### 7.2.1. プログラムの直接実行

「-x」オプションを指定してコンパイルされた opensource COBOL プログラムは、直接実行可能なプログラムとして生成される。例えば、Windows システムで「-x」オプションを指定すると「.exe」ファイルとして生成される。

これらのネイティブ実行可能ファイルは、非グラフィカルユーザインターフェースプログラムとしての実行に適している。

これは UNIX システムでは、プログラムが `bash`、`csh`、`ksh` などのコマンドシェルから実行される可能性があることを意味する。opensource COBOL プログラムが Windows システムで実行される場合、コンソールウィンドウ(つまり「`cmd.exe`」)内で実行される。

プログラムとユーザ間のやりとりは、標準入力、標準出力、および標準エラー出力を使って行われる。プログラムによって実行される画面節の入出力は、コマンドシェルの「ウィンドウ」内で実行される。

プログラムの直接実行構文は次の通りである。

**[path]program [arguments]**

例：

**/usr/local/printaccount ACCT=6625378**

または

**C:\Users\Me\Documents\Programs\printaccount.exe  
ACCT=6625378**

### 7.2.2. 「cobcrun」ユーティリティの使用

「**-m**」オプションを使用してメインプログラムに対してもコンパイラの出力形式を指定することにより、サブルーチンだけでなくすべての opensource COBOL プログラムの実行可能モジュールを生成できる(7.1.4 で説明したように、これは推奨されているサブルーチンの出力形式オプションである)。

opensource COBOL メインプログラムをこれらの動的にロード可能なモジュールにコンパイルして、「メインプログラムなのかサブルーチンなのか」を考えずに、すべてのプログラムに共通の一般的なコンパイルコマンドを使用することを好む人もいる。

この方法でコンパイルされたメインプログラムは、次のように実行する必要がある：

**[path]cobcrun program [arguments]**

プログラム名に「.so」または「.dll」拡張子を指定してはならない。「プログラム」の値は、メインプログラムの PROGRAM-ID(大文字と小文字を含む)と正確に一致する必要がある。

cobcrun の使用例：

```
cd /usr/local
cobcrun printaccount ACCT=6625378
 または
cd C:\Users\Me\Documents\Programs
cobcrun printaccount.exe ACCT=6625378
```

cobcrun コマンドでは、プログラム名でパスを指定できないことに注意が必要である。プログラムの動的ロード可能モジュールが存在するディレクトリは、現在のディレクトリであるか、現在の PATH で定義されていなければならない。

### 7.2.3. プログラムの引数

プログラムの実行方法に関係なく、プログラムに指定された引数は、6.4.2 に記載されている次のいずれかを介して取得できる。

- ACCEPT … FROM COMMAND-LINE
- ACCEPT … FROM ARGUMENT-VALUE

|                                    |                               |
|------------------------------------|-------------------------------|
| opensource COBOL Programmers Guide | opensource COBOL システムインターフェース |
|------------------------------------|-------------------------------|

#### 7.2.4. 重要な環境変数

次の表は、opensource COBOL プログラムの実行で利用できる様々な環境変数を示している。

表 7-5-実行時環境変数

| 環境変数                  | 使い方                                                                                                                                   |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| COB_LIBRARY_PATH      | opensource COBOL は実行時に、PATH およびプログラム実行可能なディレクトリから動的にロード可能なライブラリを見つけ、ロードしようとする。これらのライブラリファイルが別の場所に存在する可能性がある場合、この変数を使用してディレクトリパスを指定する。 |
| COB_PRE_LOAD          | null 以外の値に設定すると、この変数により、プログラムの実行開始時に動的ロード可能なすべてのライブラリがロードされる（モジュールを検索してロードするよりも先に）。                                                   |
| COB_SCREEN_ESC        | 空白以外の値に設定すると、この変数により ACCEPT 文が Esc キーを検出できるようになる。詳細については、表 4-8 で説明している。                                                               |
| COB_SCREEN_EXCEPTIONS | この変数を空白以外の値に設定すると、ACCEPT 文が Esc、PgUp、および PgDn キーを検出できるようになる。詳細については、表 4-8 で説明している。                                                    |
| COB_SORT_MEMORY       | この変数の値(整数)は、整列時に割り当てられるメモリ量を定義するために使用される。値が 1048576 以上の場合、「そのまま」の値がメモリ量(バイト単位)として割り当てられる。値が 1048576 未満の場合、ソ                           |



|                                    |                               |
|------------------------------------|-------------------------------|
| opensource COBOL Programmers Guide | opensource COBOL システムインターフェース |
|------------------------------------|-------------------------------|

|              |                                                                                                                                                                                                                                        |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | ートメモリ量の初期値は 128MB で設定される。                                                                                                                                                                                                              |
| COB_SWITCH_n | (n = 1~8)これらの環境変数は、SWITCH-1 から SWITCH-8 に対応する。「オン」に設定するとアクティブになり、それ以外の値はオフになる。詳細については、4.1.4 で説明している。                                                                                                                                   |
| COB_SYNC     | 大文字または小文字の「p」の値を設定すると、ファイルが書き込まれるたびにファイルを強制的にコミットする(次のコミットが発生するまでデータがメモリに保持されるのではなく、 <u>すぐに</u> ファイルに書き込まれるようにする)。これによりファイルへの更新アクセスが遅くなるが、プログラムに障害が発生した場合の整合性が向上する。                                                                    |
| DB_HOME      | opensource COBOL ビルドで Berkeley DB(BDB)パッケージを使用する場合は、この環境変数を使って、プログラムによって開かれたすべての非 SORT ファイルに関連付けられるロック管理ファイルに関連するフォルダを指定する <sup>34</sup> 。この変数を定義すると、READ 文(6.33)、REWRITE 文(6.36)、および WRITE 文(6.50)でレコードロック機能がアクティブになる <sup>35</sup> 。 |

<sup>34</sup> ORGANIZATION INDEXED ファイルでは、DB\_HOME が存在する場合、データファイルも DB\_HOME フォルダに割り当てられる。

<sup>35</sup> DB\_HOME を使用しても、Windows /MinGW 用に作成された opensource COBOL ビルドの ORGANIZATION SEQUENTIAL (いずれかのタイプ)または ORGANIZATION RELATIVE ファイルにおいてロックは機能しない。ORGANIZATION INDEXED ロックは Windows/MinGW で機能し、UNIX opensource COBOL ビルドを使ったファイル編成ではすべてのロックが機能する。

|                                    |                               |
|------------------------------------|-------------------------------|
| opensource COBOL Programmers Guide | opensource COBOL システムインターフェース |
|------------------------------------|-------------------------------|

|                                    |                                                                                                                                                                                                    |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PATH                               | opensource COBOL の「bin」ディレクトリは PATH で定義する必要がある。                                                                                                                                                    |
| TMPDIR<br>TMP<br>TEMP<br>(この順番で確認) | 一時ファイルを作成するのに適当なディレクトリ/フォルダを設定し、一時作業ファイルを作成するために SORT および MERGE によって使用される。このフォルダは、アプリケーションで必要になるどの一時ファイルに対しても使用できる。<br><br>適切な形式としては、アプリケーションが一時的な作業ファイルを作成する場合、その後でクリーンアップする必要がある <sup>36</sup> 。 |

## 7.3. 組み込みサブルーチン

### 7.3.1. 「名前による呼び出し」ルーチン

opensource COBOL には多数の組み込みサブルーチンが含まれており、一般的に Micro Focus COBOL(CBL\_...)または ACUCOBOL(C\$ ...)で使用可能なルーチンと一致することを目的としている。

これらのルーチンはすべて大文字表記で実行され、次の機能を実行することができる。

- 現在のディレクトリの変更
- ファイルのコピー
- ディレクトリの作成
- ファイルの作成、開く、閉じる、読み取り、書き込み
- ディレクトリ(フォルダ)の削除
- ファイルの削除
- サブルーチンに渡された引数の数の決定

---

<sup>36</sup> C\$DELETE および CBL\_DELETE\_FILE の組み込みサブルーチンを参照すること。

- ファイル情報の取得(サイズと最終変更日時)
- サブルーチンに渡される引数の長さ(バイト単位)の取得
- 項目の左揃え、右揃え、または中央揃えの決定
- ファイルの移動(破壊的な「コピー」)
- スリープ時間を秒単位で指定して、プログラムを「スリープ状態」にする
- スリープ時間をナノ秒単位で指定して、プログラムを「スリープ状態」にする  
警告：時間をナノ秒で表すが、Windows システムはミリ秒単位でしかスリープできない
- 実行時の opensource COBOL のバージョンに適したシェル環境にコマンドを送信する

次の表では様々な組み込みサブルーチンについて説明する。明示的に記載されている場合を除き、すべてのサブルーチン引数は必須である。値を RETURN-CODE に返すサブルーチンは、CALL 文の RETURNING / GIVING 句を利用して、選択したフルワードのバイナリ COMP-5 データ項目に結果を返すことができる。これについて 6.7 で説明している。

#### 7.3.1.1. CALL “C\$CHDIR” USING *directory-path*, *result*

このルーチンは、*directory-path* (英数字定数または一意名) を現在のディレクトリにする。

操作の戻り値は、*result* 引数 (編集されていない数値一意名) と RETURN-CODE 特殊レジスタの両方で返される。操作の戻り値は、0=成功または 128=失敗のいずれかである。

ディレクトリの変更は、プログラムが終了するまで (プログラムが再起動された場合は現在のディレクトリが自動的に復元される)、または別の C\$CHDIR が実行されるまで有効である。

#### 7.3.1.13 章—CBL\_CHANGE\_DIR を参照

### 7.3.1.2. CALL “C\$COPY” USING *src-file-path*, *dest-file-path*, 0

このサブルーチンは、「CP」(Unix) または「COPY」(Windows) コマンドを介して行われたかのように、*src-file-path* を *dest-file-path* にファイルをコピーする。

どちらのファイルパス引数も、英数字定数または一意名にすることができる。

第 3 引数は必須ではあるが、使用されない。

ファイルのコピーに失敗した場合 (例えば、ファイルまたは宛先ディレクトリが存在しない場合)、RETURN-CODE は 128 に設定され、正常に完了すると 0 に設定される。

7.3.1.16 章—CBL\_COPY\_FILE を参照

### 7.3.1.3. CALL “C\$DELETE” USING *file-path*, 0

このルーチンは、「RM」(Unix) または「ERASE」(Windows) コマンドを使用して行われたかのように、*file-path* 引数 (英数字定数または一意名) で指定されたファイルを削除する。

第 2 引数は必須ではあるが、使用されない。

ファイルの削除に失敗した場合 (例えば、ファイルが存在しない場合)、RETURN-CODE は 128 に設定され、正常に完了すると 0 に設定される。

7.3.1.20 章—CBL\_DELETE\_FILE を参照

### 7.3.1.4. CALL “C\$FILEINFO” USING *file-path*, *file-info*

このルーチンを使用すると、*file-path* 引数 (英数字定数または一意名) として指定されたファイルサイズ<sup>37</sup> と、ファイルが最後に変更された日付/時刻を取得できる。この情報は、次

---

<sup>37</sup> ファイルサイズ情報は、使用している特定の opensource COBOL ビルド/オペレーティングシステムの組み合わせでは利用できず常にゼロとして返される場合がある。

の 16 バイト領域として定義される *file-info* 引数に返される。

#### 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
```

変更時刻の小数点以下 2 桁は常に 0 である。

サブルーチンが成功すると、RETURN-CODE には 0 の値が返され、ファイルで必要な統計を取得できないと、RETURN-CODE には 35 の値が返される。2 つ未満の引数を指定すると、RETURN-CODE には 128 の値が生成される。

#### 7.3.1.14 章—CBL\_CHECK\_FILE\_EXIST を参照

#### 7.3.1.5. CALL “C\$JUSTIFY” USING data-item, “justification-type”

C\$JUSTIFY を使用して、英字、英数字、または数字の編集されたデータ項目を左、右、または中央揃えにする。 *justification-type* 引数は、実行する位置揃えのタイプを示す。その引数の値は次のように解釈される。

|         |                                   |
|---------|-----------------------------------|
| なし      | 「R」と同じように扱われる                     |
| Cxxx... | 大文字の「C」で始まる場合、値は中央揃えになる           |
| Rxxx... | 大文字の「R」で始まる場合、値は右揃えとなり、左に空白が埋められる |
| Lxxx... | 大文字の「L」で始まる場合、値は左揃えとなり、右に空白が埋められる |
| それ以外    | 「R」として扱われる                        |

#### 7.3.1.6. CALL “C\$MAKEDIR” USING *dir-path*

このルーチンを使用すると新しいディレクトリを作成でき、ディレクトリ名は、*dir-path* 引数 (英数字定数または一意名) として指定される。

指定されたパスの最下層 (最後) のディレクトリのみを作成でき、他のディレクトリは既に

存在していなければならない。このサブルーチンは、「mkdir -p」(Unix) または「mkdir /p」(Windows) としては動作しない。

RETURN-CODE は操作の戻り値に設定され、0=成功または 128=失敗のいずれかである。

7.3.1.17—CBL\_CREATE\_DIR を参照

#### 7.3.1.7. CALL “C\$NARG” USING *arg-count-result*

C\$NARG を呼び出すサブルーチンに渡された引数の数を数値項目 *arg count-result* に返す。

メインプログラムから CALL された場合、戻り値は常に 0 になる。

6.1.8 章—NUMBER-OF-CALL-PARAMETERS.register を参照

#### 7.3.1.8. CALL “C\$PARAMSIZE” USING *argument-number*

このサブルーチンは、*argument-number* パラメータ (数字定数またはデータ項目) を使用して指定されたサブルーチン引数のサイズ (バイト単位) を返す。

サイズは、RETURN-CODE 特殊レジスタに返される。

指定された引数が存在しない場合、または無効な *argument-number* が指定された場合、値には 0 が返される。

#### 7.3.1.9. CALL “C\$SLEEP” USING *seconds-to-sleep*

C\$SLEEP は、指定された秒数だけプログラムをスリープ状態にする。*seconds-to-sleep* 引数は、数字定数またはデータ項目である。

1 未満のスリープ時間は 0 として解釈され、スリープ遅延なしですぐに戻る。

7.3.1.30 章—CBL\_OC\_NANOSLEEP を参照

#### 7.3.1.10. CALL “C\$TOLOWER” USING *data-item*, BY VALUE *convert-length*

このルーチンは、*convert-length* (数字定数またはデータ項目) の *data-item* (英数字一意名) の先頭文字を小文字に変換する。

*convert-length* 引数は、**BY VALUE** で指定する必要がある。*data-item* の (先頭) 文字がいくつ変換されるかを指定し、それ以降の文字は変更されない。

*convert-length* が負またはゼロの場合、変換は実行されない。

7.3.1.35 章—CBL\_TOLOWER を参照

#### 7.3.1.11. CALL “C\$TOUPPER” USING *data-item*, BY VALUE *convert-length*

C\$TOUPPER サブルーチンは、*convert-length* (数字定数またはデータ項目) の *data-item* (英数字一意名) の先頭文字を大文字に変換する。

*convert-length* 引数は、**BY VALUE** で指定する必要がある。*data-item* の (先頭) 文字がいくつ変換されるかを指定し、それ以降の文字は変更されない。

*convert-length* が負またはゼロの場合、変換は実行されない。

7.3.1.36 章—CBL\_TOUPPER を参照

#### 7.3.1.12. CALL “CBL\_AND” USING *item-1*, *item-2*, BY VALUE *byte-length*

このサブルーチンは、ビット単位の AND 演算を項目-1 と項目-2 の左端の 8\**byte-length* の

位置同士のビットで実行し、結果のビット文字列を項目-2 に格納する。

項目-1 は英数字定数またはデータ項目で、項目-2 はデータ項目である必要がある。項目-1 と 項目-2 の長さは、少なくとも  $8 \times \text{byte-length}$  でなければならない。

*byte-length* は数字定数またはデータ項目であり、**BY VALUE** で指定する必要がある。

右の真理値表は「AND」プロセスを示している。

項目-2 の  $8 \times \text{byte-length}$  ポイントの後のビットは影響を受けない。

結果のゼロが RETURN-CODE レジスタに戻される。

| 引数 1<br>ビット | 引数 2<br>ビット | 新しい<br>引数 2<br>ビット |
|-------------|-------------|--------------------|
| 0           | 0           | 0                  |
| 0           | 1           | 0                  |
| 1           | 0           | 0                  |
| 1           | 1           | 1                  |

7.3.1.13. CALL “CBL\_CHANGE\_DIR” USING *directory-path*

このルーチンは、*directory-path* (英数字定数または一意名) を現在のディレクトリにする。

ディレクトリの変更は、プログラムが終了するまで (プログラムが再起動された場合は現在のディレクトリが自動的に復元される)、または別の CBL\_CHANGE\_DIR (または C\$CHDIR) が実行されるまで有効である。

操作の戻り値は、RETURN-CODE 特殊レジスタに返され、0=成功または 128=失敗のいずれかである。

7.3.1.1 章—C\$CHDIR を参照

7.3.1.14. CALL “CBL\_CHECK\_FILE\_EXIST” USING *file-path*, *file-info*

このルーチンは、*file-path* 引数 (英数字定数または一意名) として指定されたファイルサイ



ズ<sup>38</sup>と、ファイルが最後に変更された日付/時刻を取得できる。この情報は、次の 16 バイト領域として定義される *file-info* 引数に返される。

#### 01 Argument-2.

```

05 File-Size-In-Bytes PIC 9(18) COMP.
05 Mod-DD PIC 9(2) COMP. *> Modification Time
05 Mod-MO PIC 9(2) COMP.
05 Mod-YYYY PIC 9(4) COMP. *> Modification Date
05 Mod-HH PIC 9(2) COMP.
05 Mod-MM PIC 9(2) COMP.
05 Mod-SS PIC 9(2) COMP.
05 FILLER PIC 9(2) COMP. *> This will always be

```

00

サブルーチンが成功すると、RETURN-CODE には 0 の値が返され、ファイルで必要な統計を取得できないと、RETURN-CODE には 35 の値が返される。2 つ未満の引数を指定すると、RETURN-CODE には 128 の値が生成される。

#### 7.3.1.4 章—C\$FILEINFO を参照

#### 7.3.1.15. CALL “CBL\_CHANGE\_DIR” USING *directory-path*

CBL\_CLOSE\_FILE サブルーチンは、CBL\_OPEN\_FILE または CBL\_CREATE\_FILE サブルーチンによって既に開かれているファイルを閉じる。

*file-handle* 引数 (PIC X(4) USAGE COMP-X データ項目) によって定義されたファイルが出力用に開かれた場合、ファイルが閉じられる前に CBL\_FLUSH\_FILE が暗黙的に実行される。

サブルーチンが成功すると RETURN-CODE には 0 の値が返され、失敗すると -1 の値が返される。

---

<sup>38</sup> ファイルサイズ情報は、使用している特定の opensource COBOL ビルド/オペレーティングシステムの組み合わせでは利用できず常にゼロとして返される場合がある。

#### 7.3.1.16. CALL “CBL\_COPY\_FILE” USING *src-file-path*, *dest-file-path*

このサブルーチンは、「CP」(Unix) または「COPY」(Windows) コマンドを介して行われたかのように、*src-file-path* を *dest-file-path* にファイルをコピーする。

どちらのファイルパス引数も、英数字定数または一意名にすることができる。

ファイルのコピーに失敗した場合 (例えば、ファイルまたは宛先ディレクトリが存在しない場合)、RETURN-CODE は 128 に設定され、正常に完了すると 0 に設定される。

7.3.1.2 章—C\$COPY を参照

#### 7.3.1.17. CALL “CBL\_CREATE\_DIR” USING *dir-path*

このルーチンを使用すると新しいディレクトリを作成でき、ディレクトリ名は、*dir-path* 引数 (英数字定数または一意名) として指定される。

指定されたパスの最下層 (最後) のディレクトリのみを作成でき、他のディレクトリは既に存在していなければならない。このサブルーチンは、「mkdir -p」(Unix) または「mkdir /p」(Windows) としては動作しない。

RETURN-CODE は操作の戻り値に設定され、0=成功または 128=失敗のいずれかである。

7.3.1.6 章—C\$MAKEDIR を参照

#### 7.3.1.18. CALL “CBL\_CREATE\_FILE” USING *file-path*, 2, 0, 0, *file-handle*

CBL\_CREATE\_FILE サブルーチンは、*file-path* 引数を使用して指定された新しいファイルを作成し、CBL\_WRITE\_FILE で使用できるファイルとして出力用に開く。

引数 2、3、および 4 は、示されている定数値としてコーディングする必要がある。<sup>39</sup>

後続の **CBL\_WRITE\_FILE** または **CBL\_CLOSE\_FILE** 呼び出しに対して、*file handle*(PIC X(4) USAGE COMP-X) が返される。

サブルーチンの成功または失敗は RETURN-CODE レジスタに報告され、RETURN-CODE で-1 の値は無効な引数、0 の値は成功を示す。

#### 7.3.1.31 章—**CBL\_OPEN\_FILE** を参照

#### 7.3.1.19. CALL “**CBL\_DELETE\_DIR**” USING *dir-path*

**CBL\_DELETE\_DIR** を使って空のディレクトリを削除する。

唯一の引数—*dir-path* (英数字定数または一意名)—は、削除するディレクトリ名である。

指定したパスの最下層レベル (最後) のディレクトリのみが削除され、そのディレクトリは空でなければならない。

RETURN-CODE は操作の戻り値に設定され、0=成功または 128=失敗のいずれかである。

#### 7.3.1.20. CALL “**CBL\_DELETE\_FILE**” USING *file-path*

このルーチンは、「RM」(Unix) または「ERASE」(Windows) コマンドを使用して行われたかのように、*file-path* 引数 (英数字定数または一意名) で指定されたファイルを削除する。

ファイルの削除に失敗した場合 (例えば、ファイルが存在しない場合)、RETURN-CODE は 128 に設定され、正常に完了すると 0 に設定される。

---

<sup>39</sup> **CBL\_CREATE\_FILE** は **CBL\_OPEN\_FILE** ルーチンの特殊なケースであるため、引数 2、3、および 4 の意味について **CBL\_OPEN\_FILE** ルーチンで説明している

### 7.3.1.3 章—C\$DELETE を参照

#### 7.3.1.21. CALL “CBL\_ERROR\_PROC” USING *function, program-pointer*

このルーチンは、一般的なエラー処理ルーチンを登録する。

*function* の引数は、値が 0 または 1 の数字定数または 32 ビットのバイナリ COMP-5 データ項目（例えば USAGE BINARY-LONG）でなければならない。値 0 はエラー手続きを登録（「インストール」）、値 1 は以前にインストールされたエラー手続きを登録解除（「アンインストール」）することを意味する。

*program-pointer* は、エラー手続きのアドレスを含む USAGE PROGRAM-POINTER データ項目でなければならない。このようなデータ項目を入力する方法については、6.39.2 章で説明している。

成功 (0) または失敗 (0 以外) の結果は、RETURN-CODE レジスタに返される。

カスタムエラーハンドラルーチンがある場合は、ランタイムエラー条件が発生したときにトリガーされる。ハンドラ内のコードが実行され—EXIT PROGRAM または GOBACK が発行されると—システム標準のエラー処理ルーチンが実行される。

一度に有効にできるユーザ定義のエラー手続きは 1 つだけである。

エラー手続きはメインプログラムまたはサブプログラムによって定義できるが、登録された場所に関係なくプログラムコンパイルユニット全体に適用され、実行可能プログラムのどこかでランタイムエラーが発生したときにトリガーされる。エラー手続きがサブプログラムによって定義された場合は、エラー手続きの実行時にそのプログラムをロードする必要がある。

エラー手続きは、EXIT PROGRAM または GOBACK を使用して終了する必要がある。

以下は、エラー手続きを登録する opensource COBOL プログラムのサンプルである。プログラムの出力結果は、ご覧の通り、エラーハンドラのメッセージに続いて標準の opensource COBOL メッセージが表示される。

```
IDENTIFICATION DIVISION.
PROGRAM-ID. demoerrproc.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
78 Exit-Proc-Install VALUE 0.
01 Current-Date PIC X(8) .
01 Current-Time PIC X(8) .
01 Exit-Proc-Address USAGE PROCEDURE-POINTER.
01 Formatted-Date PIC XXXX/XX/XX.
01 Formatted-Time PIC XX/XX/XX.
PROCEDURE DIVISION.
000-Register-Err-Proc.
 SET Err-Proc-Address TO ENTRY "999-Err"
 CALL "CBL_ERROR_PROC"
 USING Err-Proc-Install, Err-Proc-Address
 END-CALL
 IF RETURN-CODE NOT = 0
 DISPLAY 'Error: Could not' &
 'register Error Procedure'
 END-IF
.
099-Now-Test-Err-Proc.
 CALL "Tilt" END-CALL
 GOBACK
.
999-Err-Proc.
 ENTRY "999-Err"
 DISPLAY
 '** A Runtime Error Has Occurred **'
 END-DISPLAY
 ACCEPT
 Current-Date FROM DATE YYYYMMDD
 END-ACCEPT
 ACCEPT
 Current-Time FROM TIME
 END-ACCEPT
 MOVE Current-Date TO Formatted-Date
 MOVE Current-Time TO Formatted-Time
 INSPECT Formatted-Time REPLACING ALL '/' BY ':'
 DISPLAY
```

プログラムの出力結果は…

```
** A Runtime Error Has Occurred **
*** 2009/08/28 10:35:10 ***
libcob: Cannot find module 'Tilt'
```

```
 '*** ' Formatted-Date ' ' Formatted-Time ' ***'
END-DISPLAY
GOBACK
.
```

### 7.3.1.22. CALL “CBL\_EXIT\_PROC” USING *function, program-pointer*

このルーチンは、一般的な終了処理ルーチンを登録する。

*function* の引数は、値が 0 または 1 の数字定数または 32 ビットのバイナリ COMP-5 データ項目（例えば USAGE BINARY-LONG）でなければならない。値 0 は終了手続きを登録（「インストール」）、値 1 は以前にインストールされた終了手続きを登録解除（「アンインストール」）することを意味する。

*program-pointer* は、終了手続きのアドレスを含む USAGE PROGRAM-POINTER データ項目でなければならない。このようなデータ項目を入力する方法については、6.39.2 章で説明している。

成功 (0) または失敗 (0 以外) の結果は、RETURN-CODE レジスタに返される。

「STOP RUN」またはそれに相当するもの（つまりメインプログラムで実行される「GOBACK」）が実行されると、終了手続きがトリガーされる。終了手続きコードが実行され、EXIT PROGRAM または GOBACK が発行されると、システム標準のプログラム終了ルーチンが実行される。

一度に有効にできるユーザ定義の終了手続きは 1 つだけである。

終了手続きはメインプログラムまたはサブプログラムによって定義できるが、登録された場所に関係なくプログラムコンパイルユニット全体に適用され、実行可能プログラムのどこかで STOP RUN が実行されたときにトリガーされる。終了手続きがサブプログラムによって定義された場合、終了手続きの実行時にそのプログラムをロードする必要がある。

終了手続きは、EXIT PROGRAM または GOBACK を使用して終了する必要がある。

以下は、終了手続きを登録する opensource COBOL プログラムのサンプルである。プログラムの出力結果も示している。

```
IDENTIFICATION DIVISION.
PROGRAM-ID. demoexitproc.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
78 Exit-Proc-Install VALUE 0.
01 Current-Date PIC X(8).
01 Current-Time PIC X(8).
01 Exit-Proc-Address USAGE PROCEDURE-POINTER.
01 Formatted-Date PIC XXXX/XX/XX.
01 Formatted-Time PIC XX/XX/XX.
PROCEDURE DIVISION.
000-Register-Exit-Proc.
 SET Exit-Proc-Address TO ENTRY "999-Exit"
 CALL "CBL_EXIT_PROC"
 USING Exit-Proc-Install, Exit-Proc-Address
 END-CALL
 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
.
999-Exit-Proc.
 ENTRY "999-Exit"
 DISPLAY
 '*** STOP RUN has been executed ***'
 END-DISPLAY
 ACCEPT
 Current-Date FROM DATE YYYYMMDD
 END-ACCEPT
 ACCEPT
 Current-Time FROM TIME
 END-ACCEPT
 MOVE Current-Date TO Formatted-Date
 MOVE Current-Time TO Formatted-Time
 INSPECT Formatted-Time REPLACING ALL '/' BY ':'
 DISPLAY
```

プログラムの出力結果は…

```
** A Runtime Error Has Occurred **
*** 2009/08/28 10:35:10 ***
libcob: Cannot find module 'Tilt'
```

```

 '*** ' Formatted-Date ' ' Formatted-Time ' ***'
END-DISPLAY
GOBACK
.

```

### 7.3.1.23. CALL “CBL\_EQ” USING *item-1*, *item-2*, BY VALUE *byte-length*

このサブルーチンは、項目-1 と項目-2 の左端の  $8 \times \text{byte-length}$  の位置同士のビットが等しいかどうか、ビット単位のテストを実行し、結果のビット文字列を項目-2 に格納する。

項目-1 は英数字定数またはデータ項目で、項目-2 はデータ項目である必要がある。項目-1 と項目-2 の長さは、少なくとも  $8 \times \text{byte-length}$  でなければならない。

*byte-length* は数字定数またはデータ項目であり、**BY VALUE** で指定する必要がある。

右の真理値表は「EQ」プロセスを示している。

項目-2 の  $8 \times \text{byte-length}$  ポイントの後のビットは影響を受けない。

結果のゼロが RETURN-CODE レジスタに戻される。

| 引数 1<br>ビット | 引数 2<br>ビット | 新しい<br>引数 2<br>ビット |
|-------------|-------------|--------------------|
| 0           | 0           | 1                  |
| 0           | 1           | 0                  |
| 1           | 0           | 0                  |
| 1           | 1           | 1                  |

### 7.3.1.24. CALL “CBL\_FLUSH\_FILE” USING *file-handle*

このサブルーチンを Micro Focus COBOL で CALL すると、*file-handle* が引数として指定された (出力) ファイルの未書込みメモリバッファがディスクに書き込まれる。

このルーチンは opensource COBOL では機能しない。Micro Focus COBOL 用に開発されたアプリケーションに互換性を提供するためだけに存在する。

### 7.3.1.25. CALL “CBL\_GET\_CURRENT\_DIR” USING BY VALUE 0, BY VALUE *length*, BY REFERENCE *buffer*

現在のディレクトリの完全修飾パス名が取得され、指定された *buffer* にパス名の *length* 文



字が保存される。

第 1 引数は使用されないが、**BY VALUE** で指定する必要がある。

*length* 引数は **BY VALUE** で指定する必要がある。

*buffer* 引数は **BY REFERENCE** で指定する必要がある。

*length* 引数 (数字定数またはデータ項目) に指定する値は、*buffer* 引数の長さを超えてはならない。

*length* 引数に指定された値が *buffer* 引数の長さよりも小さい場合、現在のディレクトリパスは左寄せされ、*buffer* の最初の *length* バイト内に空白が埋められる—そのポイント以降の *buffer* 内のバイトは変更されない。

ルーチンが成功すると、0 の値が RETURN-CODE レジスタに返される。引数(負または 0 *length* など)が原因でルーチンが失敗した場合、RETURN-CODE の値は 128 になる。第 1 引数の値がゼロ以外の場合、ルーチンは RETURN-CODE が 129 で失敗する。

#### 7.3.1.26. CALL “CBL\_IMP” USING *item-1*, *item-2*, BY VALUE *byte-length*

このサブルーチンは、ビット単位の「包含」演算を項目-1 と項目-2 の左端の  $8 * \textit{byte-length}$  の位置同士のビットで実行し、結果のビット文字列を項目-2 に格納する。

項目-1 は英数字定数またはデータ項目で、項目-2 はデータ項目である必要がある。項目-1 と項目-2 の長さは、少なくとも  $8 * \textit{byte-length}$  でなければならない。

*byte-length* は数字定数またはデータ項目であり、**BY VALUE** で指定する必要がある。

右の真理値表は「IMP」プロセスを示している。

項目-2 の 8\**byte-length* ポイントの後のビットは影響を受けない。

結果のゼロが RETURN-CODE レジスタに戻される。

| 引数 1<br>ビット | 引数 2<br>ビット | 新しい<br>引数 2<br>ビット |
|-------------|-------------|--------------------|
| 0           | 0           | 1                  |
| 0           | 1           | 1                  |
| 1           | 0           | 0                  |
| 1           | 1           | 1                  |

### 7.3.1.27. CALL “CBL\_NIMP” USING *item-1*, *item-2*, BY VALUE *byte-length*

このサブルーチンは、ビット単位の否定「包含」演算を項目-1 と項目-2 の左端の 8\**byte-length* の位置同士のビットで実行し、結果のビット文字列を項目-2 に格納する。

項目-1 は英数字定数またはデータ項目で、項目-2 はデータ項目である必要がある。項目-1 と項目-2 の長さは、少なくとも 8\**byte-length* でなければならない。

*byte-length* は数字定数またはデータ項目であり、BY VALUE で指定する必要がある。

右の真理値表は「NIMP」プロセスを示している。

項目-2 の 8\**byte-length* ポイントの後のビットは影響を受けない。

結果のゼロが RETURN-CODE レジスタに戻される。

| 引数 1<br>ビット | 引数 2<br>ビット | 新しい<br>引数 2<br>ビット |
|-------------|-------------|--------------------|
| 0           | 0           | 0                  |
| 0           | 1           | 0                  |
| 1           | 0           | 1                  |
| 1           | 1           | 0                  |

### 7.3.1.28. CALL “CBL\_NOR” USING *item-1*, *item-2*, BY VALUE *byte-length*

このサブルーチンは、ビット単位の否定 OR 演算を項目-1 と項目-2 の左端の 8\**byte-length* の位置同士のビットで実行し、結果のビット文字列を項目-2 に格納する。

項目-1 は英数字定数またはデータ項目で、項目-2 はデータ項目である必要がある。項目-1 と項目-2 の長さは、少なくとも 8\**byte-length* でなければならない。

*byte-length* は数字定数またはデータ項目であり、**BY VALUE** で指定する必要がある。

右の真理値表は「NOR」プロセスを示している。

項目-2 の  $8 * \text{byte-length}$  ポイントの後のビットは影響を受けない。

結果のゼロが RETURN-CODE レジスタに戻される。

| 引数 1<br>ビット | 引数 2<br>ビット | 新しい<br>引数 2<br>ビット |
|-------------|-------------|--------------------|
| 0           | 0           | 1                  |
| 0           | 1           | 0                  |
| 1           | 0           | 0                  |
| 1           | 1           | 0                  |

### 7.3.1.29. CALL “CBL\_NOT” USING *item-1*, BY VALUE *byte-length*

このサブルーチンは、項目-2 の左端の  $8 * \text{byte-length}$  のビットを「反転」し、結果のビット文字列を項目-2 に格納する。

項目-2 はデータ項目である必要があり、項目-2 の長さは少なくとも  $8 * \text{byte-length}$  でなければならない。

*byte-length* は数字定数またはデータ項目であり、**BY VALUE** で指定する必要がある。

右の真理値表は「NOT」プロセスを示している。

項目-2 の  $8 * \text{byte-length}$  ポイントの後のビットは影響を受けない。

結果のゼロが RETURN-CODE レジスタに戻される。

| 古い<br>引数 2<br>ビット | 新しい<br>引数 2<br>ビット |
|-------------------|--------------------|
| 0                 | 1                  |
| 1                 | 0                  |

### 7.3.1.30. CALL “CBL\_OC\_NANOSLEEP” USING *nanoseconds-to-sleep*

CBL\_OC\_NANOSLEEP は、指定されたナノ秒数だけプログラムをスリープ状態にする。

*nanoseconds-to-sleep* 引数は数字定数またはデータ項目である。

1 秒は 10 億ナノ秒であるため、プログラムを 1/4 秒間スリープさせたい場合は、*nanoseconds-to-sleep* の値に 250000000 を設定する。

#### 7.3.1.9 章—C\$SLEEP を参照

#### 7.3.1.31. CALL “CBL\_OPEN\_FILE” *file-path, access-mode, 0, 0, handle*

このルーチンは、CBL\_WRITE\_FILE または CBL\_READ\_FILE で使用できる既存のファイルを開く。

*file-path* 引数は、英数字定数またはデータ項目である。

*access-mode* 引数は、PIC X USAGE COMP-X (または USAGE BINARY-CHAR) で定義された数字定数またはデータ項目である。次のようにファイルの使用方法を指定する。

1 = 入力 (読み取り専用)

2 = 出力 (書き込み専用)

3 = 入力または出力

第 3、第 4 引数ではロックモードとデバイス仕様を指定するが、opensource COBOL には実装されていない (少なくとも現時点では)—それぞれに 0 を指定する。

最後の引数—*handle*—は PIC X(4) USAGE COMP-X 項目で、ファイルへのハンドルを受け取る。ハンドルは特定のファイルを参照するために、他のバイトストリーム関数で利用される。

RETURN-CODE -1 の値は無効な引数、0 の値は成功を示す。35 の値はファイルが存在しないことを意味する。

#### 7.3.1.18 章—CBL\_CREATE\_FILE を参照

### 7.3.1.32. CALL “CBL\_OR” USING *item-1*, *item-2*, BY VALUE *byte-length*

このサブルーチンは、ビット単位の OR 演算を項目-1 と項目-2 の左端の 8\**byte-length* の位置同士のビットで実行し、結果のビット文字列を項目-2 に格納する。

項目-1 は英数字定数またはデータ項目で、項目-2 はデータ項目である必要がある。項目-1 と項目-2 の長さは、少なくとも 8\**byte-length* でなければならない。

*byte-length* は数字定数またはデータ項目であり、BY VALUE で指定する必要がある。

右の真理値表は「OR」プロセスを示している。

項目-2 の 8\**byte-length* ポイントの後のビットは影響を受けない。

結果のゼロが RETURN-CODE レジスタに戻される。

| 引数 1<br>ビット | 引数 2<br>ビット | 新しい<br>引数 2<br>ビット |
|-------------|-------------|--------------------|
| 0           | 0           | 0                  |
| 0           | 1           | 1                  |
| 1           | 0           | 1                  |
| 1           | 1           | 1                  |

### 7.3.1.33. CALL “CBL\_READ\_FILE” USING *handle*, *offset*, *nbytes*, *flag*, *buffer*

このルーチンは、*handle* で定義されたファイルから指定された *buffer* に、バイト番号 *offset* で始まる *nbytes* のデータを読み取る。

*handle* 引数 (PIC X(4) USAGE COMP-X) は、CBL\_OPEN\_FILE への事前の呼び出しによって取り込まれている必要がある。

*offset* 引数 (PIC X(8) USAGE COMP-X) は、読み取るファイルの最初のバイト位置を定義する。ファイルの最初のバイトは、バイトオフセット 0 である。

*nbytes* 引数 (PIC X(4) USAGE COMP-X) は、読み取るバイト数(最大値)を指定する。

*flags* 引数が 128 として指定されている場合、ファイルのサイズ (バイト単位) が完了時に

ファイルオフセット引数 (引数 2) に返される。<sup>40</sup> それ以外に有効な *flags* の値は 0 だけである。この引数は、数字定数または PIC X USAGE COMP-X データ項目として指定される。

完了時に、読み取りが成功した場合は RETURN-CODE が 0 に設定され、「ファイルの終わり」条件が発生した場合は 10 に設定される。RETURN-CODE の値が -1 の場合、サブルーチン引数に問題が確認されたことを示す。

#### 7.3.1.34. CALL “CBL\_RENAME\_FILE” USING *old-file-path*, *new-file-path*

このサブルーチンを使用してファイル名を変更できる。

*old-file-path* で指定されたファイルは、*new-file-path* で指定された名前に「名前変更」される。それぞれの引数は英数字定数またはデータ項目である。

このルーチン名で気づくかもしれないが、このルーチンには単なる「名前変更」以上の機能がある—1 番目の引数に指定されたファイルを 2 番目の引数に指定されたファイルに移動する。これは、最初に *old-file-path* を *new-file-path* にコピーし、次に *old-file-path* を削除するという 2 段階の順序と考えられる。

ファイルの移動に失敗した場合 (例えば、ファイルが存在しない場合)、RETURN-CODE は 128 に設定され、正常終了すると 0 に設定される。

#### 7.3.1.35. CALL “CBL\_TOLOWER” USING *data-item*, BY VALUE *convert-length*

このルーチンは、*convert-length* (数字定数またはデータ項目) の *data-item* (英数字一意名) の先頭文字を小文字に変換する。

*convert-length* 引数は、**BY VALUE** で指定する必要がある。*data-item* の (先頭) 文字がい

---

<sup>40</sup> すべてのオペレーティングシステム/opensource COBOL 環境でファイルサイズを取得できるわけではない—そのような場合、ゼロの値が返される。

くつ変換されるかを指定し、それ以降の文字は変更されない。

*convert-length* が負またはゼロの場合、変換は実行されない。

#### 7.3.1.10 章—C\$TOLOWER を参照

#### 7.3.1.36. CALL “CBL\_Toupper” USING *data-item*, BY VALUE *convert-length*

C\$TOUPPER サブルーチンは、*convert-length* (数字定数またはデータ項目) の *data-item* (英数字一意名) の先頭文字を大文字に変換する。

*convert-length* 引数は、**BY VALUE** で指定する必要がある。*data-item* の (先頭) 文字がいくつ変換されるかを指定し、それ以降の文字は変更されない。

*convert-length* が負またはゼロの場合、変換は実行されない。

#### 7.3.1.11 章—C\$TOUPPER を参照

#### 7.3.1.37. CALL “CBL\_WRITE\_FILE” USING *handle*, *offset*, *nbytes*, 0, *buffer*

このルーチンは、指定された *buffer* から *handle* で定義されたファイルに、*nbytes* のデータをバイト番号 *offset* から書き込む。

*handle* 引数 (PIC X(4) USAGE COMP-X) は、CBL\_OPEN\_FILE への事前の呼び出しによって取り込まれている必要がある。

*offset* 引数 (PIC X(8) USAGE COMP-X) は、書き込まれるファイルの最初のバイト位置を定義する。ファイルの最初のバイトは、バイトオフセット 0 である。

*nbytes* 引数 (PIC X(4) USAGE COMP-X) は、書き込まれるバイト数(最大値)を指定する。

唯一の許容値または flags 引数は 0 である。この引数は、数字定数または PIC X USAGE COMP-X データ項目として指定される。

完了時に、書き込みが成功した場合は RETURN-CODE が 0 に設定され、I/O エラー条件が発生した場合は 30 に設定される。RETURN-CODE の値が -1 の場合、サブルーチン引数に問題が確認されたことを示す。

7.3.1.38. CALL “CBL\_XOR” USING *item-1*, *item-2*, BY VALUE *byte-length*

このサブルーチンは、ビット単位の排他的 OR 演算を項目-1 と項目-2 の左端の 8\**byte-length* の位置同士のビットで実行し、結果のビット文字列を項目-2 に格納する。

項目-1 は英数字定数またはデータ項目で、項目-2 はデータ項目である必要がある。項目-1 と項目-2 の長さは、少なくとも 8\**byte-length* でなければならない。

*byte-length* は数字定数またはデータ項目であり、BY VALUE で指定する必要がある。

右の真理値表は「XOR」プロセスを示している。

項目-2 の 8\**byte-length* ポイントの後のビットは影響を受けない。

結果のゼロが RETURN-CODE レジスタに戻される。

| 引数 1<br>ビット | 引数 2<br>ビット | 新しい<br>引数 2<br>ビット |
|-------------|-------------|--------------------|
| 0           | 0           | 0                  |
| 0           | 1           | 1                  |
| 1           | 0           | 1                  |
| 1           | 1           | 0                  |

7.3.1.39. CALL “SYSTEM” USING *command*

このサブルーチンは、指定された *command* (英数字定数またはデータ項目) をコマンドシェルに送信する。

CALL を SYSTEM に発行する opensource COBOL プログラムに従属するシェルが開かれる。



|                                    |                               |
|------------------------------------|-------------------------------|
| opensource COBOL Programmers Guide | opensource COBOL システムインターフェース |
|------------------------------------|-------------------------------|

コマンドからの出力 (コマンドが存在する場合) は、opensource COBOL プログラムが実行されたコマンドウィンドウに表示される。

Unix システムでは、シェル環境は標準のシェルプログラムを使用して構築される。これは、Cygwin Unix エミュレータで作成された opensource COBOL ビルドを使用する場合も同様である。

ネイティブ Windows Windows/MinGW ビルドでは、シェル環境は使用している Windows のバージョンに適した Windows コンソールウィンドウコマンドプロセッサ (通常は「cmd.exe」) となる。

実行されたコマンドからの出力をトラップして opensource COBOL プログラム内で処理するには、パイプ (>) を使用してコマンド出力を一時ファイルに送信し、制御が戻ったらプログラム内から読み取る。

## 8. サンプルプログラム

### 8.1. FileStat-Msgs.cpy – ファイル状態コード

このコピーブックには、ファイル I/O 文によって生成されるであろう 2 桁のファイル状態コードを変換するための EVALUATE 文が含まれている。

コピーブックでは、ファイル状態データ項目の名前が「STATUS」で、エラーメッセージデータ項目の名前が「MSG」であると想定している。ただし、COPY 文の REPLACING 句を使用すると、次のようにユーザが名付けたデータ名を扱うことができる。

```
COPY FileStat-Msgs
REPLACING STATUS BY Input-File-Status
MSG BY Error-Message.
```

以下は、コピーブック「FileStat-Msgs.cpy」である。

```
EVALUATE STATUS
WHEN 00 MOVE 'SUCCESS' TO MSG
WHEN 02 MOVE 'SUCCESS DUPLICATE' TO MSG
WHEN 04 MOVE 'SUCCESS INCOMPLETE' TO MSG
WHEN 05 MOVE 'SUCCESS OPTIONAL' TO MSG
WHEN 07 MOVE 'SUCCESS NO UNIT' TO MSG
WHEN 10 MOVE 'END OF FILE' TO MSG
WHEN 14 MOVE 'OUT OF KEY RANGE' TO MSG
WHEN 21 MOVE 'KEY INVALID' TO MSG
WHEN 22 MOVE 'KEY EXISTS' TO MSG
WHEN 23 MOVE 'KEY NOT EXISTS' TO MSG
WHEN 30 MOVE 'PERMANENT ERROR' TO MSG
WHEN 31 MOVE 'INCONSISTENT FILENAME' TO MSG
WHEN 34 MOVE 'BOUNDARY VIOLATION' TO MSG
WHEN 35 MOVE 'FILE NOT FOUND' TO MSG
WHEN 37 MOVE 'PERMISSION DENIED' TO MSG
WHEN 38 MOVE 'CLOSED WITH LOCK' TO MSG
WHEN 39 MOVE 'CONFLICT ATTRIBUTE' TO MSG
WHEN 41 MOVE 'ALREADY OPEN' TO MSG
WHEN 42 MOVE 'NOT OPEN' TO MSG
WHEN 43 MOVE 'READ NOT DONE' TO MSG
WHEN 44 MOVE 'RECORD OVERFLOW' TO MSG
WHEN 46 MOVE 'READ ERROR' TO MSG
WHEN 47 MOVE 'INPUT DENIED' TO MSG
WHEN 48 MOVE 'OUTPUT DENIED' TO MSG
```

```

 WHEN 49 MOVE 'I/O DENIED' ' TO MSG
 WHEN 51 MOVE 'RECORD LOCKED' ' TO MSG
 WHEN 52 MOVE 'END-OF-PAGE' ' TO MSG
 WHEN 57 MOVE 'I/O LINAGE' ' TO MSG
 WHEN 61 MOVE 'FILE SHARING FAILURE' ' TO MSG
 WHEN 91 MOVE 'FILE NOT AVAILABLE' ' TO MSG
END-EVALUATE.

```

## 8.2. COBDUMP -16 進数/文字データダンプサブルーチン

次のサンプルプログラムは、渡されたデータ域の書式設定された16進数と文字のダンプを生成するための、便利で小さなユーティリティサブルーチンである。

opensource COBOL フォーラムをフォローしている場合は、CBL\_OC\_DUMP サブルーチンが opensource COBOL プログラミング大会で優勝したことを聞いたことがあるだろう。これはデータダンプを作成するための優れたツールであり、近いうちに公式の opensource COBOL ディストリビューションに含まれるだろう。

```

IDENTIFICATION DIVISION.
PROGRAM-ID. COBDUMP.

** This is an opensource COBOL subroutine that will generate a
**
** formatted Hex/Char dump of a storage area. To use this **
** subroutine, simply CALL it as follows: **
** **
** CALL "COBDUMP" USING <data-item> **
** [<length>] **
** **
** If specified, the <length> argument specifies how many **
** bytes of <data-item> are to be dumped. If absent, all of **
** <data-item> will be dumped (i.e. LENGTH(<data-item>) will **
** be assumed for <length>). **
** **
** >>> Note that the subroutine name MUST be specified in <<< **
** >>> UPPERCASE <<< **
** **
** The dump is generated to STDERR, so you may pipe it to a **
** file when you execute your program using "2> file". **
** **

```

```

** AUTHOR: GARY L. CUTLER **
** CutlerGL@gmail.com **
** **
** NOTE: The author has a sentimental attachment to **
** this subroutine - it's been around since 1971 **
** and it's been converted to and run on 10 dif- **
** ferent operating system/compiler environments **
** **
** DATE-WRITTEN: October 14, 1971 **
** **

** DATE CHANGE DESCRIPTION **
** =====
** GC1071 Initial coding - Univac Dept. of Defense COBOL '68 **
** GC0577 Converted to Univac ASCII COBOL (ACOB) - COBOL '74 **
** GC1182 Converted to Univac UTS4000 COBOL - COBOL '74 w/ **
** SCREEN SECTION enhancements **
** GC0883 Converted to Honeywell/Bull COBOL - COBOL '74 **
** GC0983 Converted to IBM VS COBOL - COBOL '74 **
** GC0887 Converted to IBM VS COBOL II - COBOL '85 **
** GC1294 Converted to Micro Focus COBOL V3.0 - COBOL '85 w/ **
** extensions **
** GC0703 Converted to Unisys Universal Compiling System (UCS) **
** COBOL (UCOB) - COBOL '85 **
** GC1204 Converted to Unisys Object COBOL (OCOB) - COBOL 2002 **
** GC0609 Converted to opensource COBOL 1.1 - COBOL '85 w/ some
COBOL **
** 2002 features **
** GC0410 Enhanced to make 2nd argument (buffer length) **
** optional **

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
 FUNCTION ALL INTRINSIC.
DATA DIVISION.
WORKING-STORAGE SECTION.
78 Undisplayable-Char-Symbol VALUE X'F9'.
01 Addr-Pointer USAGE POINTER.
01 Addr-Number REDEFINES Addr-Pointer
 USAGE BINARY-LONG.

01 Addr-Sub USAGE BINARY-CHAR.

01 Addr-Value USAGE BINARY-LONG.

01 Buffer-Length USAGE BINARY-LONG.

```

```

01 Buffer-Sub COMP-5 PIC 9(4) .

01 Hex-Digits VALUE '0123456789ABCDEF' .
05 Hex-Digit OCCURS 16 TIMES PIC X(1) .

01 Left-Nibble COMP-5 PIC 9(1) .
01 Nibble REDEFINES Left-Nibble
 BINARY-CHAR .

01 Output-Detail .
05 OD-Addr .
10 OD-Addr-Hex OCCURS 8 TIMES PIC X .
05 FILLER PIC X(1) .
05 OD-Byte PIC Z(3)9 .
05 FILLER PIC X(1) .
05 OD-Hex OCCURS 16 TIMES .
10 OD-Hex-1 PIC X .
10 OD-Hex-2 PIC X .
10 FILLER PIC X .
05 OD-ASCII OCCURS 16 TIMES
 PIC X .

01 Output-Sub COMP-5 PIC 9(2) .

01 Output-Header-1 .
05 FILLER PIC X(80) VALUE
 '<-Addr-> Byte ' &
 '<----- Hexadecimal -----> ' &
 '<---- Char ---->' .

01 Output-Header-2 .
05 FILLER PIC X(80) VALUE
 '==== ' &
 '===== ' &
 '==== ' .

01 PIC-XX .
05 FILLER PIC X VALUE LOW-VALUES .
05 PIC-X PIC X .
01 PIC-Halfword REDEFINES PIC-XX
 PIC 9(4) COMP-X .

01 PIC-X10 .
05 FILLER PIC X(2) .
05 PIC-X8 PIC X(8) .

01 Right-Nibble COMP-5 PIC 9(1) .

```

LINKAGE SECTION.

01 Buffer PIC X ANY LENGTH.

01 Buffer-Len USAGE BINARY-LONG.

PROCEDURE DIVISION USING Buffer, OPTIONAL Buffer-Len.  
000-COBDUMP.

```

 IF NUMBER-OF-CALL-PARAMETERS = 1
 MOVE LENGTH(Buffer) TO Buffer-Length
 ELSE
 MOVE Buffer-Len TO Buffer-Length
 END-IF
 MOVE SPACES TO Output-Detail
 SET Addr-Pointer TO ADDRESS OF Buffer
 PERFORM 100-Generate-Address
 MOVE 0 TO Output-Sub
 DISPLAY
 Output-Header-1 UPON SYSERR
 END-DISPLAY
 DISPLAY
 Output-Header-2 UPON SYSERR
 END-DISPLAY
 PERFORM VARYING Buffer-Sub FROM 1 BY 1
 UNTIL Buffer-Sub > Buffer-Length
 ADD 1
 TO Output-Sub
 END-ADD
 IF Output-Sub = 1
 MOVE Buffer-Sub TO OD-Byte
 END-IF
 MOVE Buffer (Buffer-Sub : 1) TO PIC-X
 IF (PIC-X < ' ')
 OR (PIC-X > '~')
 MOVE Undisplayable-Char-Symbol
 TO OD-ASCII (Output-Sub)
 ELSE
 MOVE PIC-X
 TO OD-ASCII (Output-Sub)
 END-IF
 DIVIDE PIC-Halfword BY 16
 GIVING Left-Nibble
 REMAINDER Right-Nibble
 END-DIVIDE
 ADD 1 TO Left-Nibble
 Right-Nibble
 END-ADD
 MOVE Hex-Digit (Left-Nibble)
 TO OD-Hex-1 (Output-Sub)

```

```
 MOVE Hex-Digit (Right-Nibble)
 TO OD-Hex-2 (Output-Sub)
 IF Output-Sub = 16
 DISPLAY
 Output-Detail UPON SYSERR
 END-DISPLAY
 MOVE SPACES TO Output-Detail
 MOVE 0 TO Output-Sub
 SET Addr-Pointer UP BY 16
 PERFORM 100-Generate-Address
 END-IF
 END-PERFORM
 IF Output-Sub > 0
 DISPLAY
 Output-Detail UPON SYSERR
 END-DISPLAY
 END-IF
 EXIT PROGRAM
.
100-Generate-Address.
 MOVE 8 TO Addr-Sub
 MOVE Addr-Number TO Addr-Value
 MOVE ALL '0' TO OD-Addr
 PERFORM WITH TEST BEFORE UNTIL Addr-Value = 0
 DIVIDE Addr-Value BY 16
 GIVING Addr-Value
 REMAINDER Nibble
 END-DIVIDE
 ADD 1 TO Nibble
 MOVE Hex-Digit (Nibble)
 TO OD-Addr-Hex (Addr-Sub)
 SUBTRACT 1 FROM Addr-Sub
 END-PERFORM
.
```

## opensource COBOL Programmer's Guide

### 【制作】

OSS コンソーシアム オープン COBOL ソリューション部会

### 【原著】

Gary Cutler ("OpenCOBOL 1.1 Programmer's Guide")

### 【翻訳・執筆】

東京システムハウス株式会社 島田桃花

### 【監修】

東京システムハウス株式会社 比毛寛之、上野俊作、井坂徳恭  
株式会社 SIT11 飯島裕一

### 【協力】(50 音順)

OVOL ICT ソリューションズ株式会社  
株式会社 SIT11  
株式会社 CIJ  
サン情報サービス株式会社

### 【発行】

OSS コンソーシアム オープン COBOL ソリューション部会  
URL: <https://www.osscons.jp/osscobol/>