opensource COBOL Programmer's Guide

3rd Edition, 18 September 2024

翻訳 OSS コンソーシアム
オープン COBOL ソリューション部会

改訂履歴

版	発行日	改訂詳細
初版 v1.0.0	2023/8/31	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。
v1.1.0	2023/9/21	誤字や翻訳漏れを修正。
v1.2.0	2023/10/16	条件名の訳語を一部修正。
v2.0.0	2024/2/29	通貨記号の既定値を「＄」から「￥」に変更。
		「1.4. ソースコードの形式」に$IF、$ELSE、$ENDに関する記述を追加。
		「1.6. COPY文の使い方」にREPLACING句のLEADING/TRAILING指定、JOINING句のPREFIX/SUFFIX指定、PREFIXING/SUFFIXING句に関する記述を追加。
		「1.7. 定数の使い方 」に「1.7.3. 日本語定数」を追加。
		「4.2.1. ファイル管理段落 図4-10-ファイル管理段落構文」を一部修正。
		「4.2.1.3. 索引編成ファイル」に分割キーに関する記述を追加。
		「5.3. データ記述の形式」にASCENDING KEY/DESCENDING KEY句とINDEXED BY句の記述順の許容に関する記述を追加。
		「6.1.4.2.5. 比較条件 図6-12-比較条件構文」を一部修正。
		「6.8. CANCEL」に「6.8.2. CANCEL文の書き方2 — CANCEL ALL」を追加。
		「6.13. DELETE」に「6.13.2. DELETE文の書き方2 — DELETE FILE」を追加。
		新しい章として「7. 日本語対応」を追加。
		「8.1.2. コンパイルオプション」に「-assign_external」と「-free_1col_aster」の項目を追加。
		「8.1.7. 重要な環境変数 表8-4-環境変数コンパイラ」に環境変数「COB_DATE」「COB_IO_ASSUME_REWRITE」「COB_NIBBLE_C_UNSIGNED」「COB_VERBOSE」「OC_EXTEND_CREATES」「OC_IO_CREATES」「OC_USERFH」の項目を追加。
		「8.3.1.「名前による呼び出し」ルーチン 」に「8.3.1.1. CALL “C$CALLEDBY” USING program-name GIVING status」「8.3.1.7. CALL “C$LIST-DIRECTORY” USING item-1, item-2, item-3」「8.3.1.32. CALL “CBL_OC_KEISEN” USING item-1」を追加。
v3.0.0	2024/9/18	日本語翻訳マニュアル HTML版を公開。
v3.1.0	2025/2/25	訳語を一部修正。
		「4.2. 入出力節 図4-9-入出力節構文」を一部修正。
		「5.3. データ記述の形式 表5-9-数字編集PICTURE記号」を一部修正。
		「6.1.4.2.5. 比較条件 図 6-12-比較条件構文」を一部修正。
v3.2.0	2025/3/31	日本語翻訳マニュアル HTML版からPDF版を作成するよう変更。

v3.3.0	2025/5/26	構文画像の表示方法をHTMLテーブルに変更。
		「6.28.2. MOVE文の書き方2 — MOVE CORRESPONDING」を一部修正。
		「8.2.4. 重要な環境変数」を一部修正。
		「8.1.2. コンパイルオプション」を一部修正。

以上

目次

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MinGW [1] 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キング処理など、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。

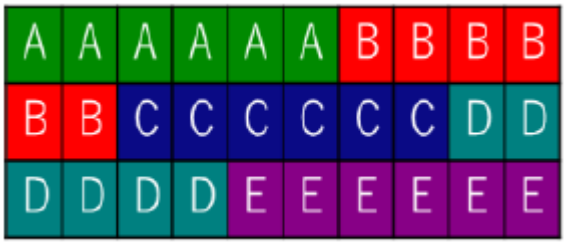
1.2.3. 構造化データ

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。

記述方法	説明
ORGANIZATION IS LINE SEQUENTIAL	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。 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、純粋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、純粋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まったく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た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まったく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プログラムで使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(6.1.8.2)と共有(6.1.8.1)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。

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に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つの両極端の中間に位置します。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 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 または LOCALE-DATE組み込み関数
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 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 C$TOLOWER組み込みサブルーチン(8.3.1.12) CBL_TOLOWER組み込みサブルーチン(8.3.1.38)
文字列を大文字に変換する	UPPER-CASE組み込み関数 C$TOUPPER組み込みサブルーチン(8.3.1.13) CBL_TOUPPER組み込みサブルーチン(8.3.1.39)
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 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「$12,342.19-」を「-12342.19」という値に復号する)	NUMVAL組み込み関数 NUMVAL-C組み込み関数
文字列または文字列を格納できるデータ項目の長さを決定する	LENGTH組み込み関数 または BYTE-LENGTH組み込み関数
文字列の開始位置と長さに基づいて部分文字列を抽出する	「送信」項目に部分参照を含むMOVE文(6.28.1)
桁区切り記号 (日本では「,」)、通貨記号(日本では「￥」)、小数点、クレジット/デビット記号、先頭または末尾の記号文字を含む、出力用の数値項目を書式化する	受け取り項目に適用されたPICTURE編集記号(5.3)を指定したMOVE文(6.28)
文字列項目の位置揃え(左、右、または中央)	C$JUSTIFY組み込みサブルーチン(8.3.1.6)
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) SUBSTITUTE組み込み関数

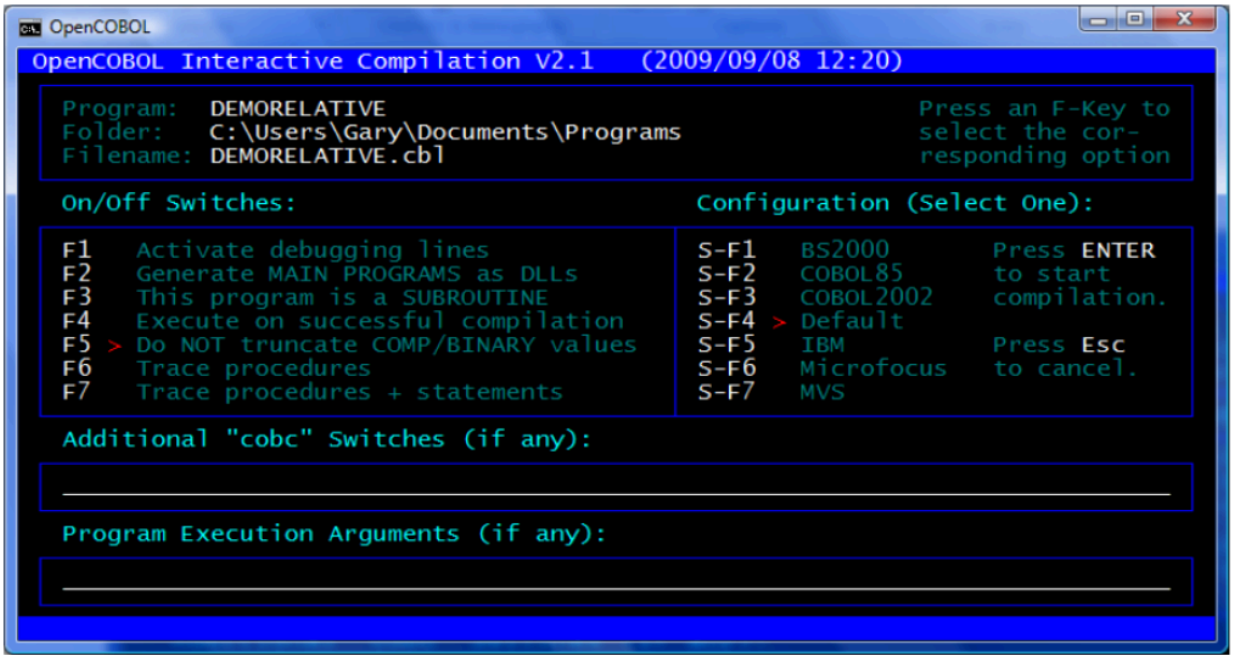
	および SUBSTITUTE-CASE組み込み関数
文字列を解析し、1つ以上の区切り文字順序に基づいて部分文字列に分割する これらの区切り文字は、単一の文字、複数の文字列、またはいずれかが重複した連続オカレンスの可能性がある	UNSTRING文(6.49)
文字列の先頭または末尾の空白の削除	TRIM組み込み関数
部分文字列の開始文字位置と長さに基づいて、単一の部分文字列を同じ長さの別の部分文字列に置換する	「受け取り」項目に部分参照を含むMOVE文(6.28.1)
文字列内にある1つ以上の部分文字列を、オカレンス位置に関係なく、同じ長さの置換部分文字列に置換する	REPLACINGオプション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

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。

構文	説明
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、セレクター、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。

記述方法	説明
-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、ソースコード入力が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。

記述方法	説明
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。
7桁目の「$IF」	ソース行が有効なopensource COBOLコードであり、opensource COBOLコンパイラに「–fdebugging-line」スイッチが指定されていない限り(その場合、行はコンパイルされる)コメントであることを示す。固定形式モードの場合のみ有効である。
7桁目の「$IF」	$IF 文の書き方 1 $IF 定数名-1 [NOT] { < / > / = } 定数-1 定数名-1がコンパイルオプション「-constant」で指定されており、定数-1の値が定数名-1の値に等しい時、または定数名-1の値の範囲内にある時、$IF文以降に続くソース行の処理が実行される。 $IF 文の書き方 2 $IF 定数名-2 [NOT] DEFINED 定数名-2がコンパイルオプション「-constant」で指定されている時、DEFINED句は真となり、$IF文以降に続くソース行の処理が実行される。それ以外の場合はNOT DEFINED句が真となり、$IF文以降に続くソース行の処理が実行される。
7桁目の「$ELSE」	直前の$IF文の条件式が偽である時、$ELSE文に制御が移り、$ELSE文以降に続くソース行の処理が実行される。直前の$IF文の条件式が真である時、$ELSE文は無視される。
7桁目の「$END」	$END文と同じレベルにある$IF文または$ELSE文に続くソース行の処理の実行が終了すると、$ENDに制御が移り、$IF文または$ELSE文の終了を示す。
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(,)またはセミコロン(;)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コンパイラー(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：

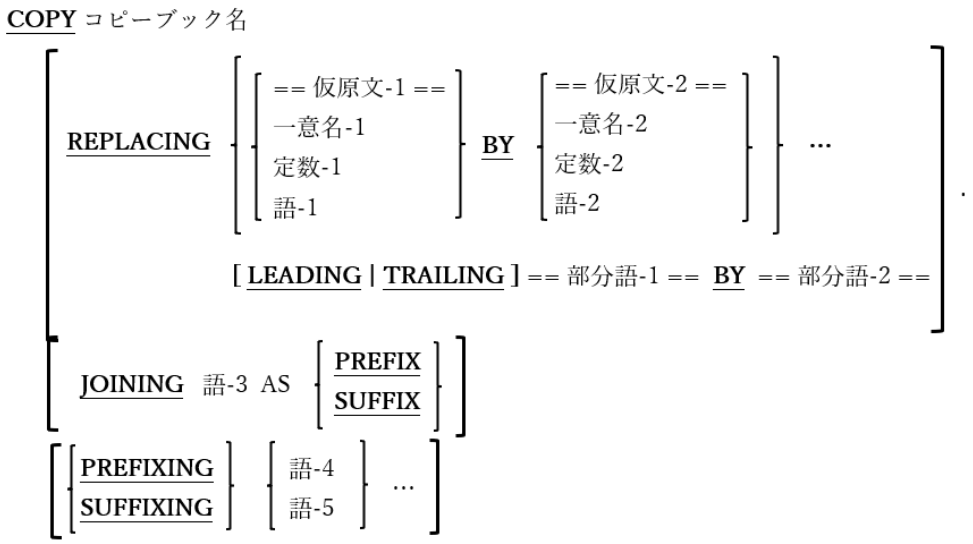
```
CALL “SUBROUTINE” USING 1,2
```

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」の可能性をなくすためにセミコロン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。

1. 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。
2. 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。
3. 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。

4. 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。

5. 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。

6. オプションのREPLACING句の使用時にLEADINGおよびTRAILINGを指定すると、予約語またはユーザ定義語の先頭(LEADINGを参照)または末尾(TRAILINGを参照)にある文字順序を置き換えることができる。例えば、「0100-xxxxxx」の単語を「020-xxxxxx」に変換するには、LEADING ==0100-== BY ==020-== とコーディングする。「0100-」の単語を削除するには、LEADING ==0100-== BY ==== とコーディングする。

7. オプションのJOINING句の使用時にPREFIXを指定すると、コピーブックで定義されているデータ名、条件名、定数名の前に語-3とハイフン「-」が付けられる。

8. オプションのJOINING句の使用時にSUFFIXを指定すると、コピーブックで定義されているデータ名、条件名、定数名の後ろにハイフン「-」と語-3が付けられる。

9. オプションのPREFIXING句を使用すると、コピーブックで定義されているデータ名、条件名、定数名の前に語-4が付けられる。

10. オプションのSUFFIXING句を使用すると、コピーブックで定義されているデータ名、条件名、定数名の後ろに語-5が付けられる。

11. 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、「8.1.8 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”($1F_{16}=31_{10}$)、h’22’($22_{16}=34_{10}$)、H’DEAD’($DEAD_{16}=57005_{10}$)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。

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”($4A4B4C_{16}$=ASCII文字列「JKL」)、x’20’(20_{16}=空白)、X’30313233’(30313233_{16}=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(1.4)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に使用した方)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
12345678901234567890123456789012345678901234567890123456789012345678901234567890
       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
12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

```
       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。

1.7.3. 日本語定数

- N”日本語”、n’あいう’などは日本語定数を構成する。「N」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。N以外にも「NC」や「ND」が使用できる。
- NX’E38184E3828DE381AF’(E38184E3828DE381AF$_{16}$=SHIFT-JIS文字列「いろは」)などの16進数定数。「NX」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進数の日本語定数は、各文字が16ビット分のデータ(4桁の16進数)で表されるため、常に4の倍数の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。

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	0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。

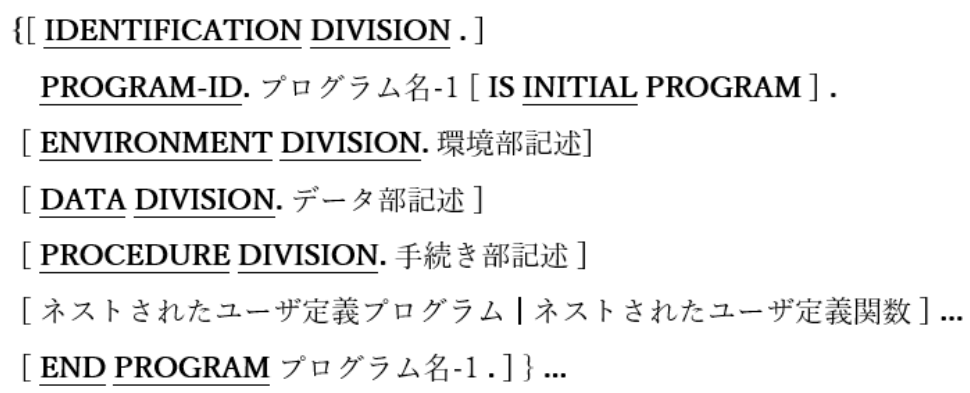
```
01 Demo-Identifier PIC X(27). *> This is a 27-character data-item
.
.
.
DISPLAY LENGTH OF “This is a LENGTH OF Example”
DISPLAY LENGTH OF Demo-Identifier
DISPLAY 27
```

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



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。

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。

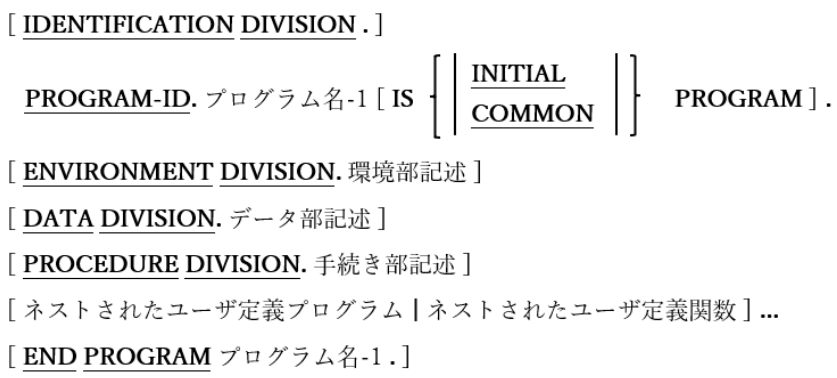
4. 各区分の目的の概要は次の通りである：

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



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。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

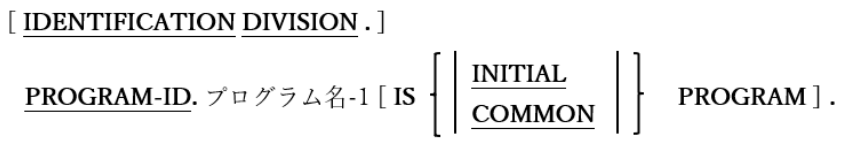
ユーザ定義関数はopensource COBOLの構文として定義されているが、現在はサポートされていない。

1. ユーザ定義関数をコンパイルしようとすると、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



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

```
CONFIGRATION 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 ] .
```

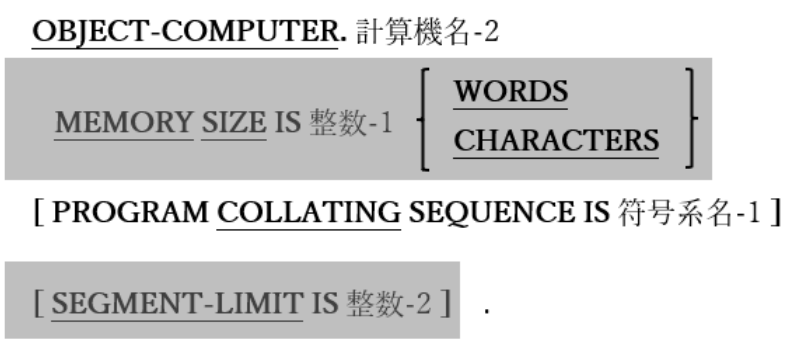
翻訳計算機(SOURCE-COMPUTER)段落は、プログラムがコンパイルされる計算機を定義する。

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。

3. 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1.4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

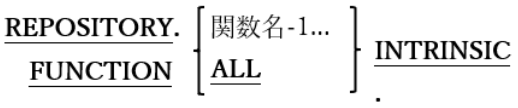


実行用計算機(OBJECT-COMPUTER)段落は、プログラムが実行される計算機について説明する段落ではあるが、単なるドキュメントではない。

1. 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。
2. 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。
3. 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

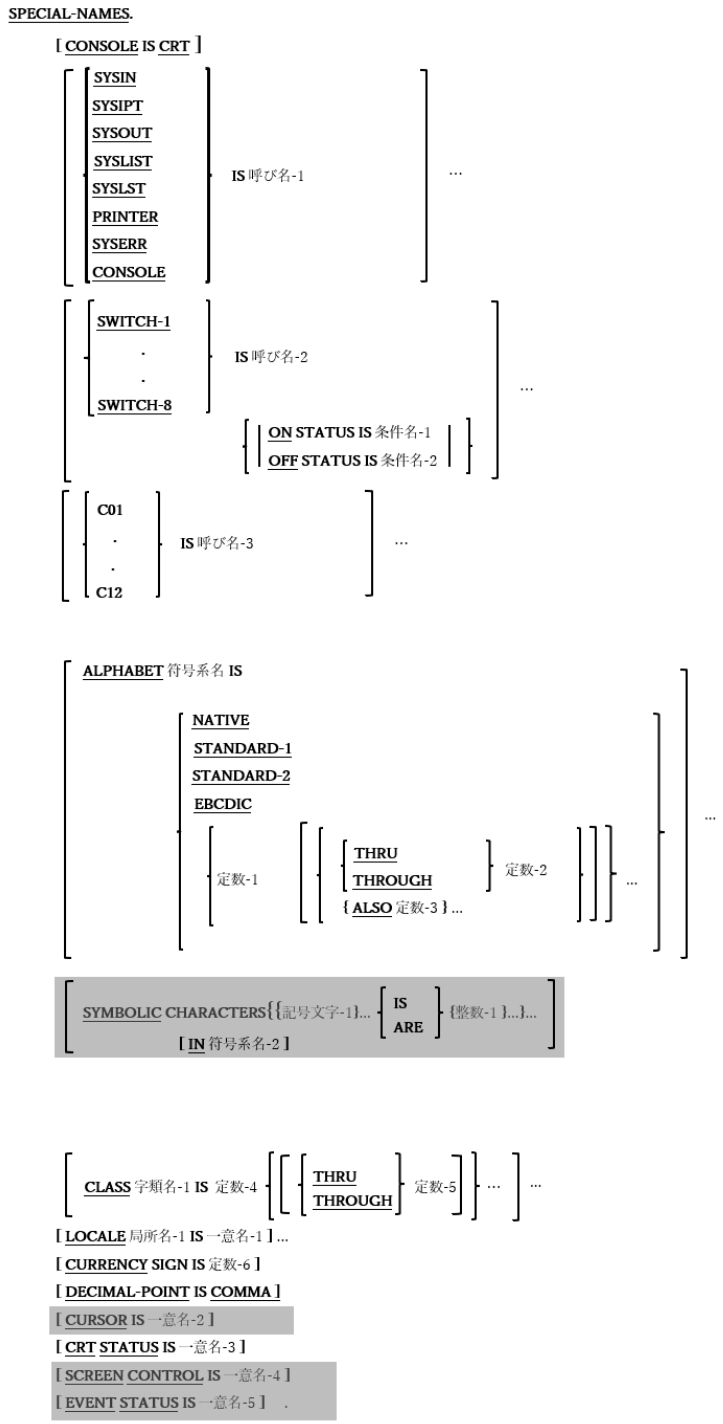


リポジトリ(REPOSITORY)段落は、様々な組み込み関数へのアクセスを制御するためのメカニズムを定義する。

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。
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



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呼び名の関連付け、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 呼び名-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編集記号で使用され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円記号(¥)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(1.5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

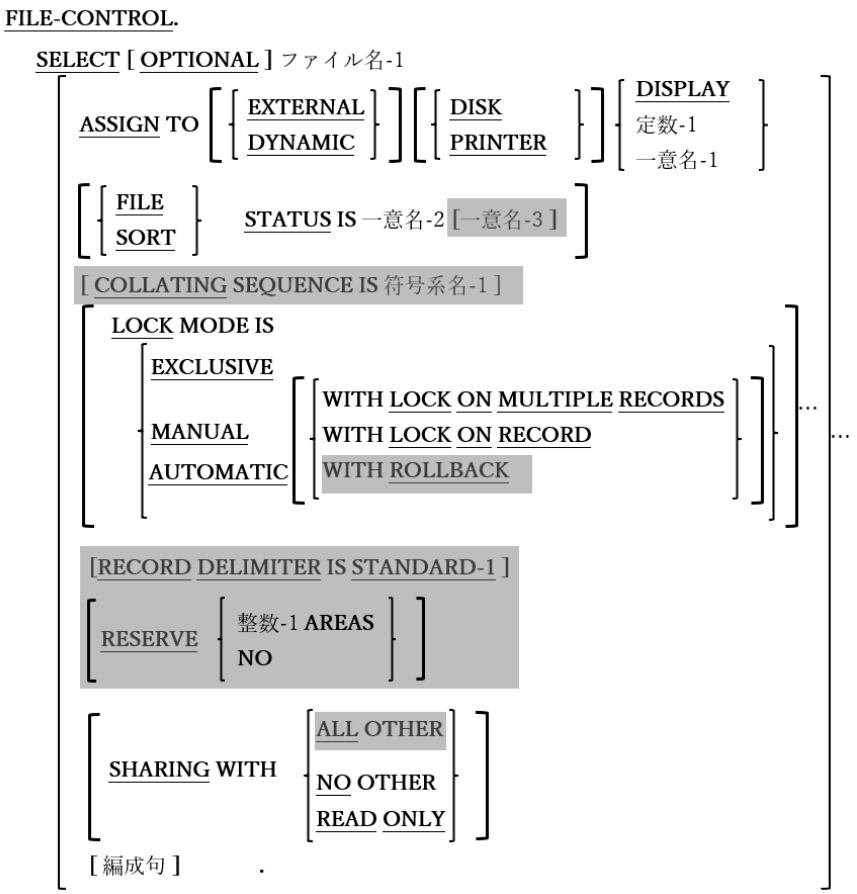
```
INPUT-OUTPUT SECTION.
[ FILE-CONTROL. ファイル管理記述 ]
[ I-O-CONTROL. 入出力管理記述 ]
```

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8.1.8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8.1.8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

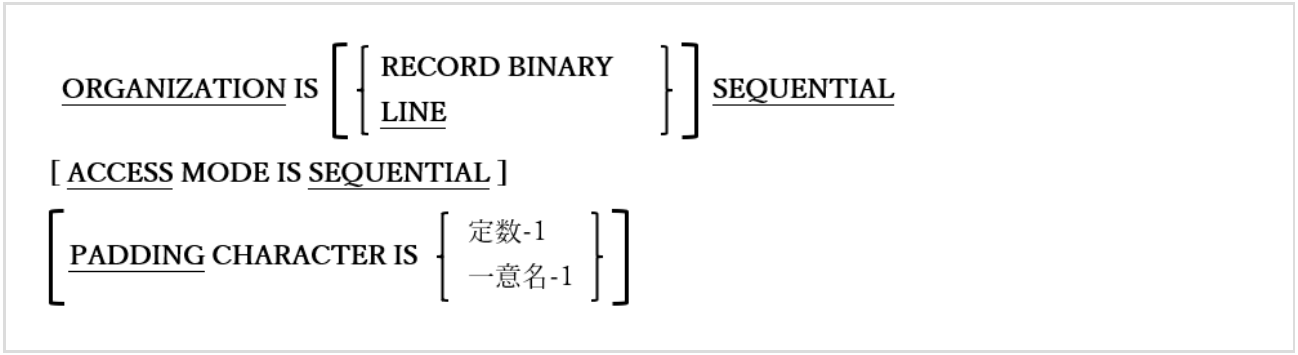
ステータス値	意味
00	成功
02	成功(重複レコードキーが検出された)
05	成功(オプションファイルが存在しない)
07	成功(ユニットが存在しない)
10	ファイル終了
14	キー範囲外

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
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、6.1.8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

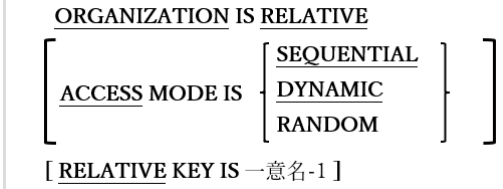


SEQUENTIALファイルとは、ファイル内のデータを順次処理することしかできない内部構造(COBOLでは編成と呼ばれる)を持つファイルである。ファイルの100番目のレコードを読み取るには、レコードの1から始めて99までを読み取る必要がある。

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

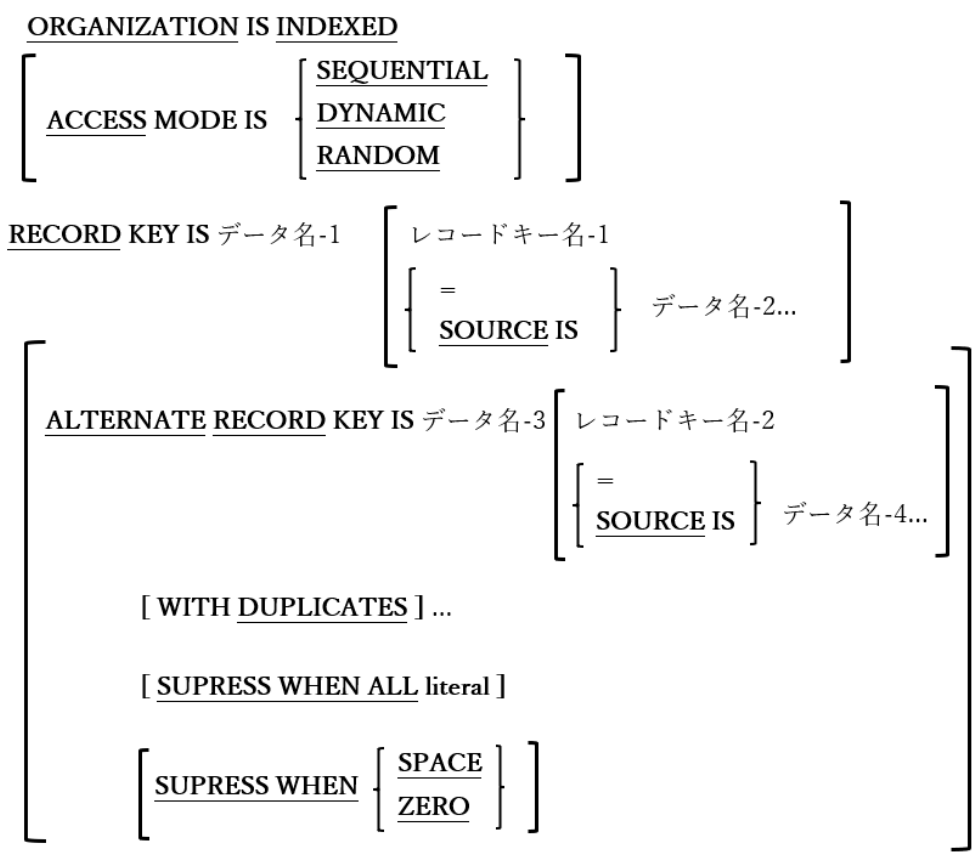


RELATIVEファイルは、レコードを順次またはランダムに処理できる内部編成を持つファイルであり、ファイル内の相対レコード番号を指定することによって、レコードの読み取り、書き込み、および更新を行うことができる。

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. 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)。

2. 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。

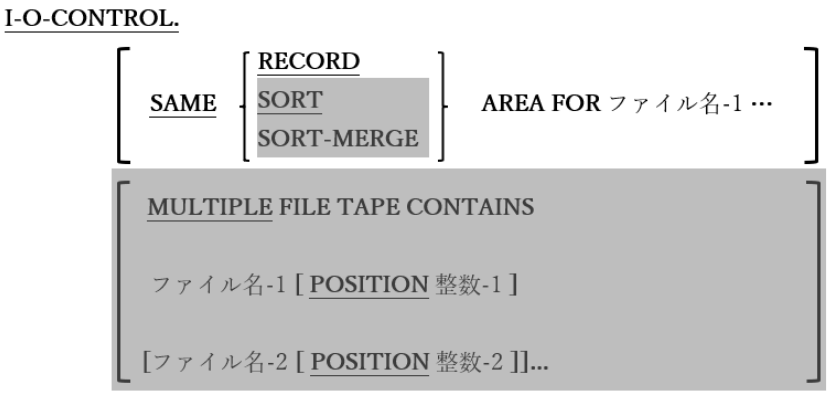
3. RECORD KEY句は、ファイル内レコードへ一次アクセスするために用いるレコード内の項目を定義する。この時、ファイル内の2つのレコードが同じPRIMARY KEY項目値を持つことは許可されない。SOURCE IS句は、分割キーで使用する。

4. 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。

5. 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。
6. 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。
7. 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

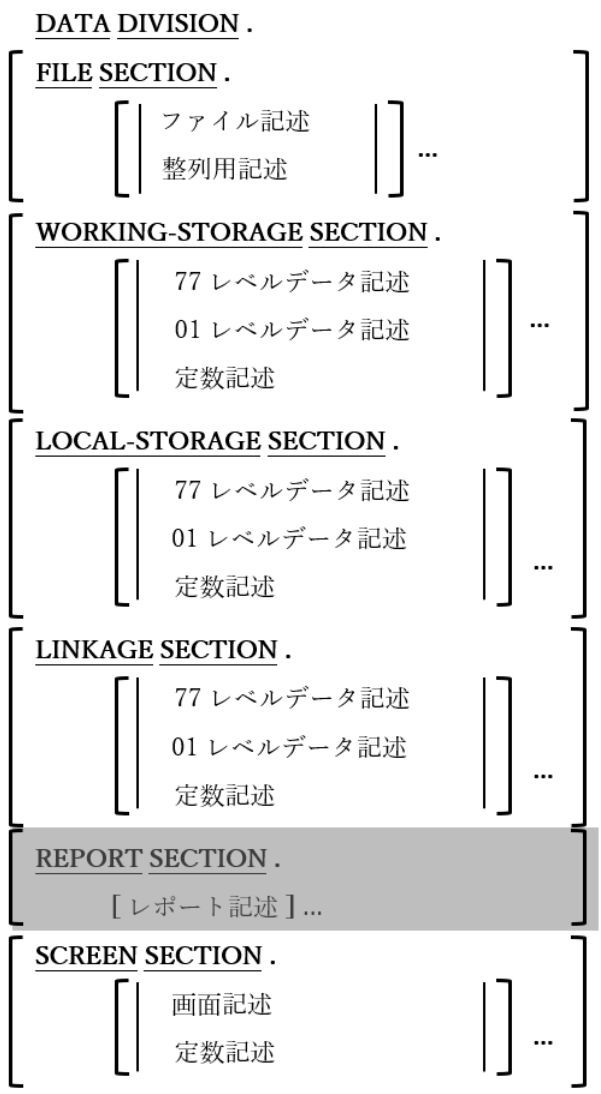


入出力管理(I-O-CONTROL)段落は、特定のファイル処理を最適化するために用いる。

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、上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。

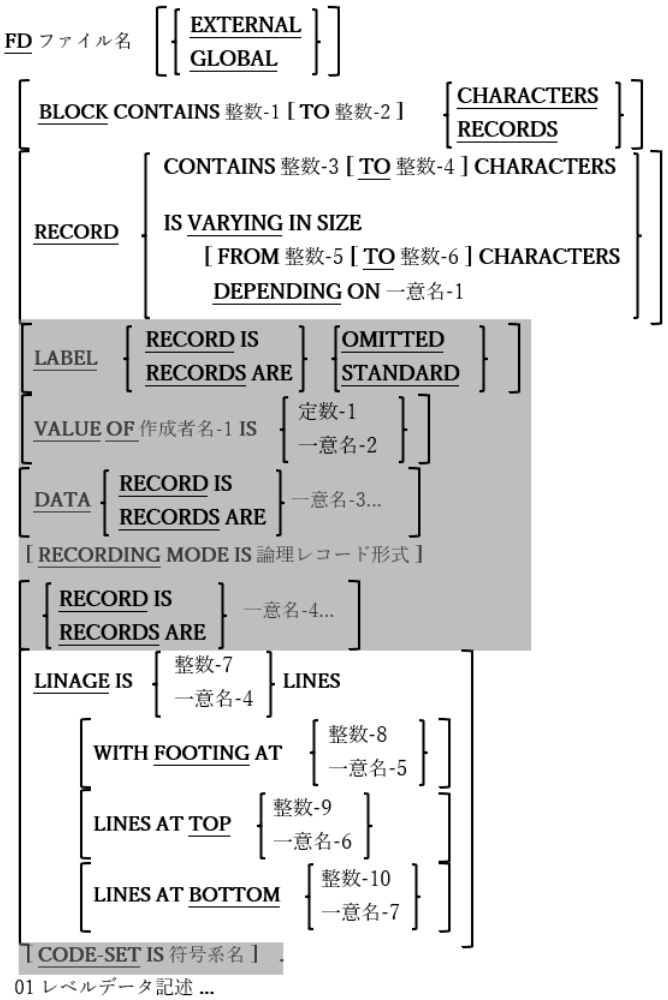
4. 局所場所はネストされたプログラムでは使用できない。

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、ファイルで使用されるレコード形式と、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、バッファー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。

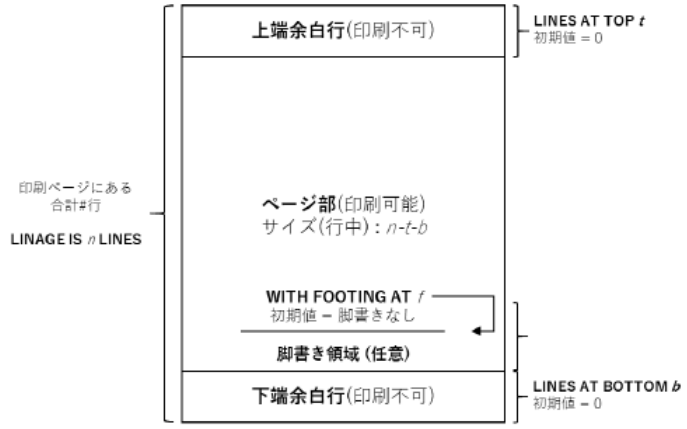
4. LINE SEQUENTIALファイルを使用する場合、RECORD CONTAINS句とRECORD IS VARYING句は無視される(警告メッセージが表示される)。他のファイル編成において、これらのような相互に排他的な句は、ファイル内のデータレコードの長さを定義していて、その長さはブロックのサイズを計算するためにBLOCK CONTAINS ･･･ RECORDS句によって使用される。

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 SEQUENTIA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



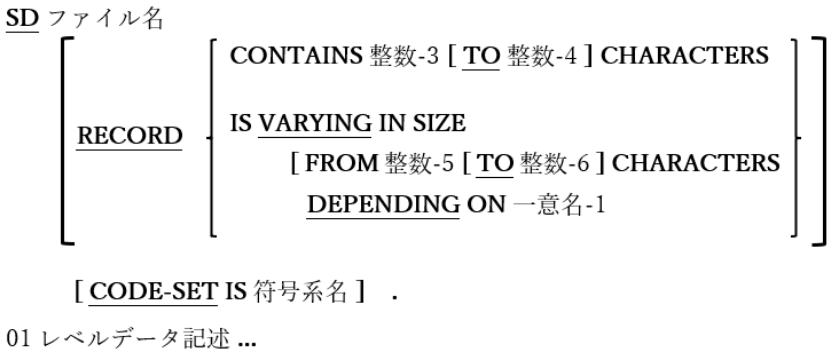
8. EXTERNAL句を指定することにより、ファイル記述が必要な各コンパイルユニットで(EXTERNAL句を使って)記述されている場合、ファイル記述は、特定の実行スレッド内のすべてのプログラム(個別にコンパイルされるか、同じコンパイルユニットでコンパイルされる)間で共有できる。この共有によって、異なる様々なプログラムでファイルをOPEN、読み書き、CLOSEすることができる。

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

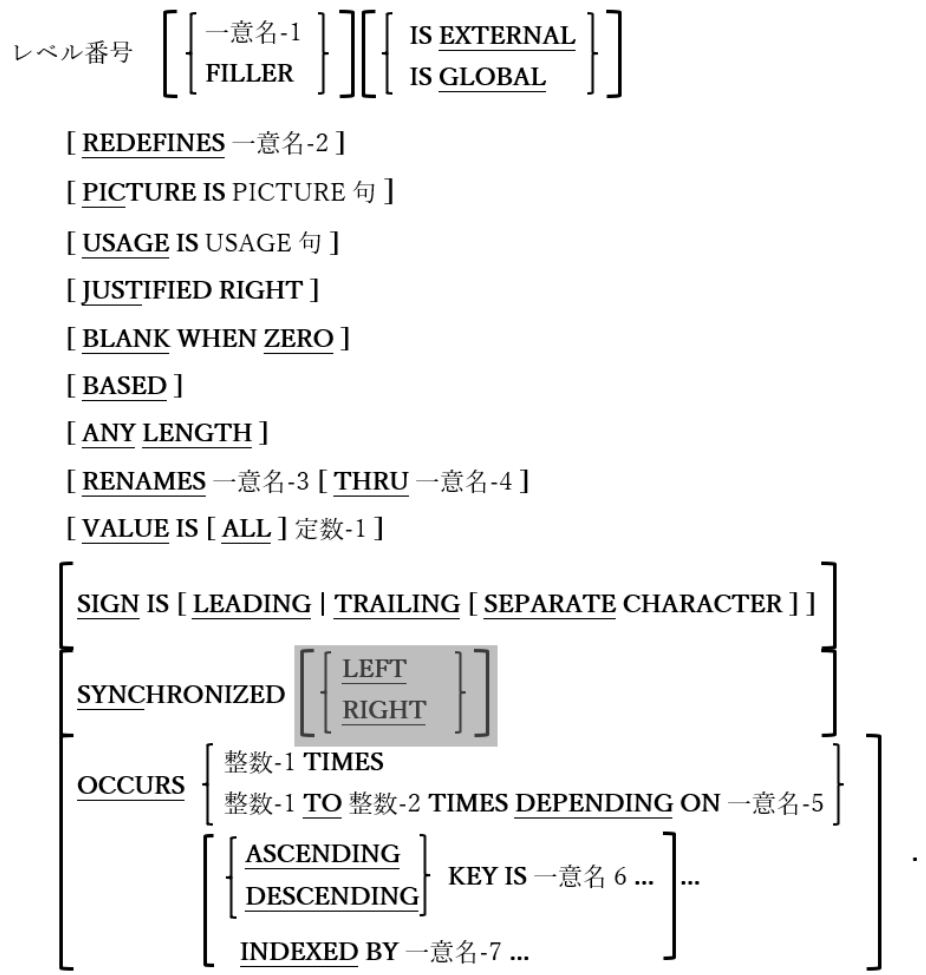


整列用ファイル(6.27および6.40.1を参照)はファイル記述ではなく、整列用記述を使って説明する。

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(8.2.4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。

レベル番号	説明
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つの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。
N	1つの日本語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「N」は2バイトのストレージをす。

以上の四つの記号は、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：

PICTURE句	**説明**
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 - 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記号で使用できる。
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。

PIC N(10)	10文字の日本語文字を格納できるデータ項目で、20バイトのストレージが割り当てられる。

「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」データ項目で使用でき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	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。
	このようなデータ項目に対して実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。
	値がそのよ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。
	例えば、会社の今年の総収益に何百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：
	`01 Gross-Revenue PIC 9(9).`
	ことき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。
	`01 Gross-revenuePIC 9(3)P(6).`
	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。
	`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り、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(LEARDING)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)	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。 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。 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、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。 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： 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。 例(記号b̶は空白を表す)：
￥[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。

数値	PICTURE 句	結果
17	-999	b̶017
-17	-999	-017
265	-----99	b̶b̶b̶b̶265
-265	-----99	b̶b̶b̶-265
51	999-	051b̶
-51	999-	051-

<table>
<tr><td></td><td>「￥」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<s>b</s>は空白を表す)：
<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>265</td><td>¥¥¥¥¥99</td><td><s>bbb</s>¥265</td></tr></table></td></tr>
<tr><td>*(アスタリスク)</td><td>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：
<table><tr><th>数値</th><th>PICTURE 句</th><th>結果</th></tr><tr><td>265</td><td>*****99</td><td>****265</td></tr></table></td></tr>
<tr><td>,(カンマ)[8]</td><td>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<s>b</s>は空白を表す)：
<table><tr><th>数値</th><th>PICTURE 句</th><th>結果</th></tr><tr><td>17</td><td>¥¥,¥¥¥,¥99</td><td><s>bbbbbbb</s>¥17</td></tr><tr><td>265</td><td>¥¥,¥¥¥,¥99</td><td><s>bbbbbb</s>¥265</td></tr><tr><td>1456</td><td>¥¥,¥¥¥,¥99</td><td><s>bbbb</s>¥1,456</td></tr></table></td></tr>
<tr><td>.(ピリオド)[8]</td><td>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！
例：
<code>01 Edited-Value PIC 9(3).99.</code></td></tr>
</table>

	`01 Payment PIC 9(3)V99 VALUE 152.19.` `...` `MOVE Payment TO Edited-Value. DISPLAY Edited-Value.` 152.19が表示される。

/(スラッシュ)	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。
	英数字編集項目の場合、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。
	日本語編集項目の場合、挿入された「／」文字は、編集結果で2バイトのストレージを占有する。
	例：
	`01 Edited-Date PIC 99/99/9999.`
	`...`
	`MOVE 08182009 TO Edited-Date.`
	`DISPLAY Edited-Date.`
	08/18/2009が表示される。

+(プラス)	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。 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。 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、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。 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： 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「＋」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 -17 / +999 / -017 265 / +++++99 / ␣␣␣+265 -265 / +++++99 / ␣␣␣-265 51 / 999+ / 051+ -51 / 999- / 051-

0(ゼロ)	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。
	例：
	`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。

<table>
<tr><td>B</td><td>この記号は、空白文字を編集値に挿入する。
英数字編集項目の場合、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。
日本語編集項目の場合、挿入された日本語空白文字は、編集結果で2バイトのストレージを占有する。
例：
<code>01 Edited-Phone-Number PIC 9(3)B9(3)B9(4).</code>
<code>...</code>
<code>MOVE 5185551212 TO Edited-Phone-Number.</code>
<code>DISPLAY Edited-Phone-Number.</code>
518 555 1212と表示される。</td></tr>
<tr><td>CR</td><td>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。
1つのPICTURE句で複数の「CR」記号を使用することはできない。
「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。
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。
例(記号<s>b</s>は空白を表す)：
<table><tr><th>数値</th><th>PICTURE 句</th><th>結果</th></tr><tr><td>17</td><td>99CR</td><td>17<s>bb</s></td></tr><tr><td>-17</td><td>99CR</td><td>17CR</td></tr></table></td></tr>
<tr><td>DB</td><td>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。
1つのPICTURE句で複数の「DB」記号を使用することはできない。
「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。
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。
例(記号<s>b</s>は空白を表す)：
<table><tr><th>数値</th><th>PICTURE 句</th><th>結果</th></tr><tr><td>17</td><td>99DB</td><td>17bb</td></tr><tr><td>-17</td><td>99DB</td><td>17DB</td></tr></table></td></tr>
<tr><td>Z</td><td>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「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。
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。
編集プロセスは、次のように機能すると考えること：
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。</td></tr>
</table>

例(記号␣は空白を表す)：

数値	PICTURE 句	結果
17	Z999	␣017
265	ZZZZZ99	␣␣␣␣265

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。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、6.38(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。

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
BINARY	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(8.1.9)の「バイナリサイズ」設定によって異なる。	最互換性—26項参照	PICTURE句に「S」記号がある場合は可	可	COMPUTATIONAL, COMPUTATIONAL-4
BINARY-CHAR or BINARY-CHAR SIGNED	1バイト	ネイティブ—26項参照	可	不可	
BINARY-CHAR UNSIGNED	1バイト	ネイティブ—26項参照	不可—27項参照	不可	
BINARY-C-LONG or BINARY-C-LONG SIGNED	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—26項参照	可	不可	
BINARY-C-LONG UNSIGNED	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—26項参照	不可—27項参照	不可	
BINARY-DOUBLE or BINARY-DOUBLE SIGNED	「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—26項参照	可	不可	
BINARY-DOUBLE UNSIGNED	「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—26項参照	不可 —27項参照	不可	

BINARY-LONG or BINARY-LONG SIGNED	ワード(32ビット)のストレージを割り当てる。	ネイティブ — 26項参照	可	不可	SIGNED-LONG, SIGNED-INT
BINARY-LONG UNSIGNED	ワード(32ビット)のストレージを割り当てる。	ネイティブ — 26項参照	不可 — 27項参照	不可	UNSIGNED-LONG, UNSIGNED-INT
BINARY-SHORT or BINARY-SHORT SIGNED	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 — 26項参照	可	不可	SIGNED-SHORT
BINARY-SHORT UNSIGNED	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 — 26項参照	不可 — 27項参照	不可	UNSIGNED-SHORT
COMPUTATIONAL	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(8.1.8)の「バイナリサイズ」設定によって異なる。	最互換性 — 26項参照	PICTURE句に「S」記号がある場合は可	可	BINARY, COMPUTATIONAL-4
COMPUTATIONAL-1	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	
COMPUTATIONAL-2	「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	
COMPUTATIONAL-3	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(29項参照)	パック10進数 — 28項参照	PICTURE句に「S」記号がある場合は可	不可	PACKED-DECIMAL

COMPUTATIONAL-4	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(8.1.8)の「バイナリサイズ」設定によって異なる。	最互換性 — 26項参照	PICTURE句に「S」記号がある場合は可	可	BINARY, COMPUTATIONAL
COMPUTATIONAL-5	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(8.1.8)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	
COMPUTATIONAL-X	プログラムのコンパイルに使用される構成ファイル内の「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、8.1.8を参照すること。	最互換性 — 26項参照	PICTURE句に「S」記号がある場合は可	可	
DISPLAY	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10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	
INDEX	ワード(32ビット)のストレージを割り当てる。	ネイティブ — 26項参照	不可	不可	
NATIONAL	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。				
PACKED-DECIMAL	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(29項参照)	パック10進数 — 28項参照	PICTURE句に「S」記号がある場合は可	不可	COMPUTATIONAL-3

POINTER	ワード(32ビット)のストレージを割り当てる。	ネイティブ — 26項参照	不可	不可	
PROGRAM-POINTER	ワード(32ビット)のストレージを割り当てる。	ネイティブ — 26項参照	不可	不可	
SIGNED-INT	ワード(32ビット)のストレージを割り当てる。	ネイティブ — 26項参照	可	不可	BINARY-LONG-SIGNED, SIGNED-LONG
SIGNED-LONG	ワード(32ビット)のストレージを割り当てる。	ネイティブ — 26項参照	可	不可	BINARY-LONG SIGNED, SIGNED-INT
SIGNED-SHORT	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 — 26項参照	可	不可	BINARY SHORT SIGNED
UNSIGNED-INT	ワード(32ビット)のストレージを割り当てる。	ネイティブ — 26項参照	不可 — 27項参照	不可	BINARY-LONG UNSIGNED, UNSIGNED-LONG
UNSIGNED-LONG	ワード(32ビット)のストレージを割り当てる。	ネイティブ — 26項参照	不可 — 27項参照	不可	BINARY-LONG UNSIGNED, UNSIGNED-INT
UNSIGNED-SHORT	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 — 26項参照	不可 — 27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と、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と、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。

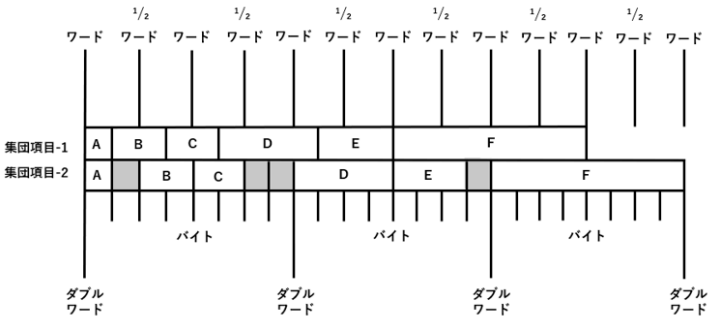
    c. バイナリ項目が4バイトのストレージを占有する場合、バイナリ項目は次のワード境界に割り当てられる。

    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

```
01  Group-Item-1.                          01  Group-Item-2.
    05 A          PIC X(1).                  05 A            PIC X(1).
    05 B         USAGE BINARY-SHORT.       05 B    SYNC    USAGE BINARY-SHORT.
    05 C          PIC X(2).                  05 C            PIC X(2).
    05 D          USAGE BINARY-LONG.        05 D    SYNC    USAGE BINARY-LONG.
    05 E          PIC X(3).                  05 E            PIC X(3).
    05 F            USAGE BINARY-DOUBLE.      05 F     SYNC     USAGE BINARY-
DOUBLE.
```



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分からないが、必要であれば使用できる。

---

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。

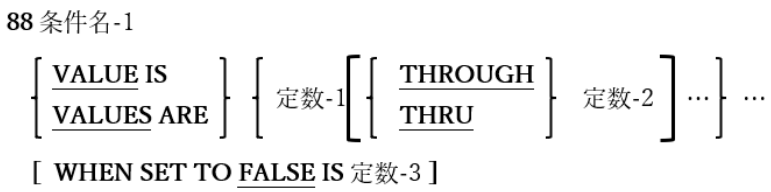
8 特殊名段落でDECIMAL-POINT IS COMMAが指定されている場合、「.」と「,」の意味と使い方が反転する。

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。

10 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。

5.4. 条件名

図5-12-レベル88条件名記述構文

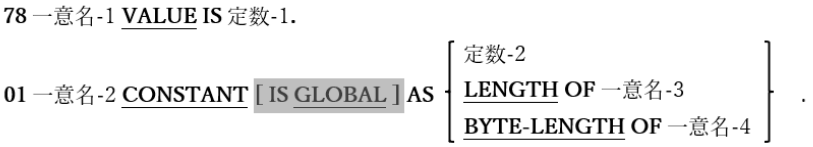


条件名はブーリアン型(つまり「TRUE」/「FALSE」)のデータ項目である。

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

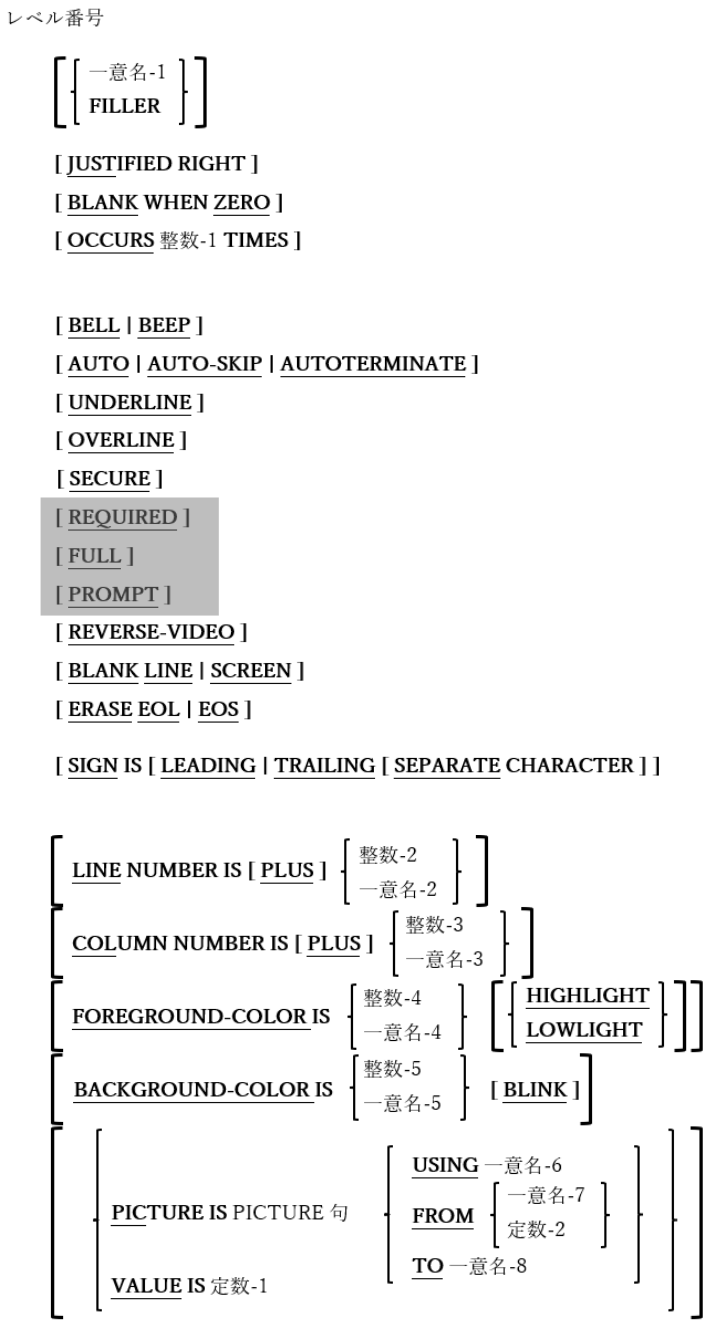


この形式のデータ項目は、実際にストレージを割り当てることはないが、その代わりに、名前を英数字または数字定数に関連付ける役割がある。

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



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。 11

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。

---

11 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)。

6. 手続き部

6.1. 構成要素

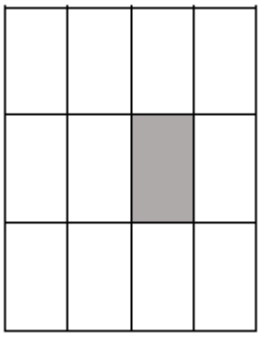
6.1.1. 表の参照

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.
```

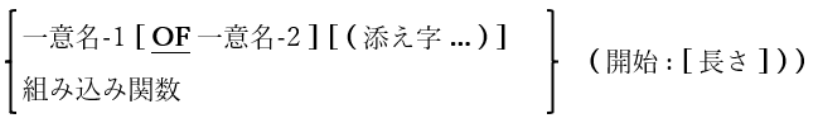
テーブルへの参照を修飾する場合は次のように記述する。

```
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残り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。

例	説明
CUSTOMER-LAST-NAME (1:3)	CUSTOMER-LAST-NAMEの最初の3文字を参照する。
CUSTOMER-LAST-NAME (4:)	CUSTOMER-LAST-NAMEの4番目以降のすべての文字位置を参照する。
FUNCTION CURRENT-DATE (5:2)	現在の月を参照する。
Hex-Digits (Nibble + 1:1)	「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)項目であるとすると、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

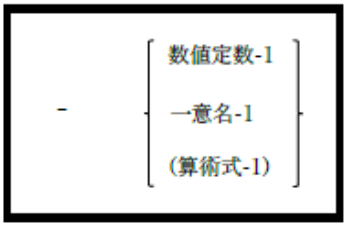
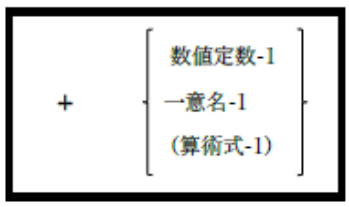
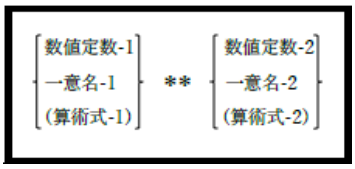
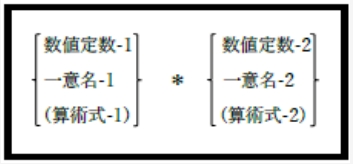
opensource COBOLは他のCOBOL実装と同様に、基本となる2つの式をサポートする。

- 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 演算子	意味
図6-2-符号(-) 1番目(最上位) 	単項減算演算子(-)は引数の算術否定を返す。引数と数字定数の-1を掛けた値を有効値とする。
図6-3-符号(+) 1番目(最上位) 	単項加算演算子(+)は引数の値を返す。引数と数字定数の+1を掛けた値を有効値とする。
図6-4-べき乗演算子 2番目 	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。
図6-5-乗算演算子 3番目 	演算子の左右の引数の乗算を求める。
図6-6-除算演算子 3番目	演算子の左右の引数の除算を求める。

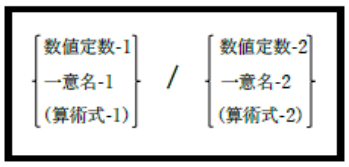
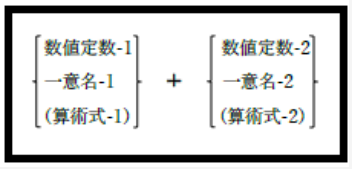
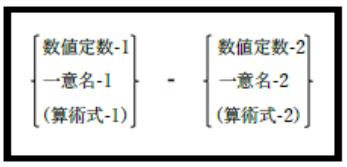


図6-7-加算演算子 4番目(最下位) 	演算子の左右の引数の加算を求める。
図6-8-減算演算子 4番目(最下位) 	左側の引数から右側の引数を引いた値を求める。

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)。

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「HUMONGOUS」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。

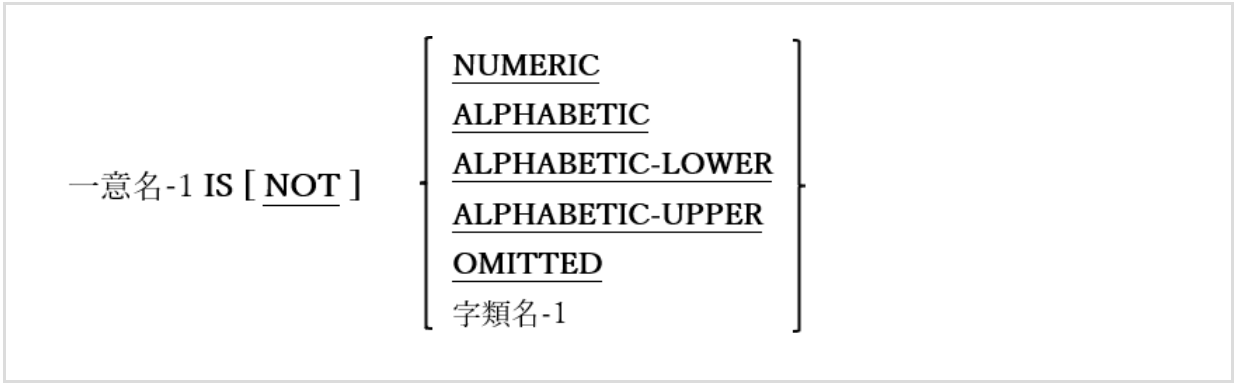
```
IF SHIRT-SIZE = 17 OR SHIRT-SIZE = 17.5
```

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

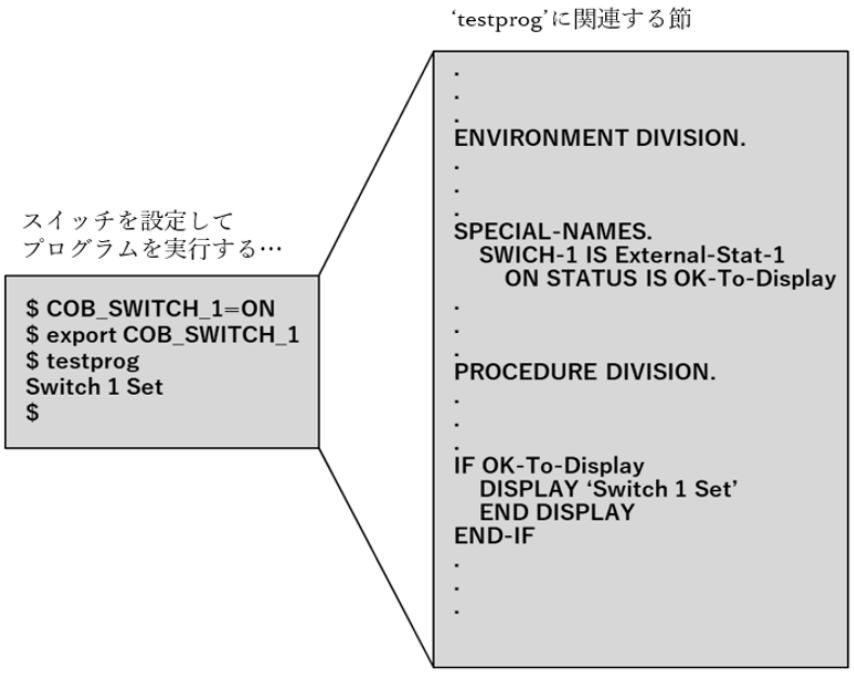
```
一意名-1 IS [ NOT ] { POSITIVE
                    { NEGATIVE
                    { ZERO
```

正負条件は、PIC 9データ項目の数値状態を判別する。

1. 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。
2. POSITIVEまたはNEGATIVE字類条件は一意名-1の値がそれぞれ0より大きいか小さい場合、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

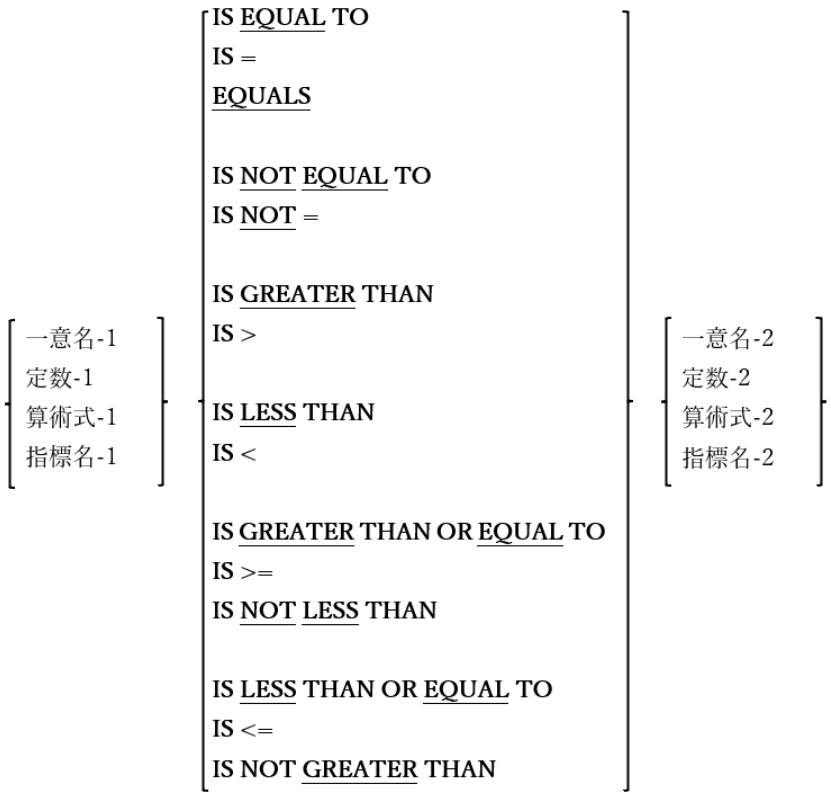
図6-11-スイッチ状態条件



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

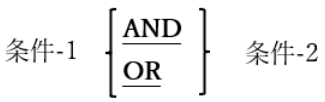


比較条件では、2つの異なる値がどのように「比較」し合っているかを判別する。

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)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。

   ```
   FALSE OR TRUE AND TRUE          結果：TRUE
   (FALSE OR FALSE) AND TRUE       結果：FALSE
   FALSE OR (FALSE AND TRUE)       結果：TRUE
   ```

6.1.4.2.7. 否定条件

図6-14-否定条件構文

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
DD 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 `12` 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".
```

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、「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。 13

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。

---

12 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。

13 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？

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 PIC/USAGE句[14]	使用方法
LINAGE-COUNTER	BINARY-LONG 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)。 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)。 ***このレジスタの内容は変更してはいけない。***
NUMBER-OF-CALL-PARAMETERS	BINARY-LONG SIGNED	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。 同じデータを取得する別の方法については、8.3.1.9のC$NARG組み込みサブルーチンのドキュメントを参照。
RETURN-CODE	BINARY-LONG SIGNED	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、それを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。ほとんどの組み込みサブルーチン(8.3)が、このレジスタを使用して値を返す。 これらの値は—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。
SORT-RETURN	BINARY-LONG SIGNED	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。
WHEN-COMPILED	See “Usage”	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。

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

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 [15] 、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 [16] 、「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。

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。

共有オプション	効果
ALL 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 OTHER	あなたのプログラムがファイルを使用している限り、他のどんなプログラムによる、どんな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 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。

---

[15] 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。

[16] 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。

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—他の同時実行プログラムによる—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。

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。

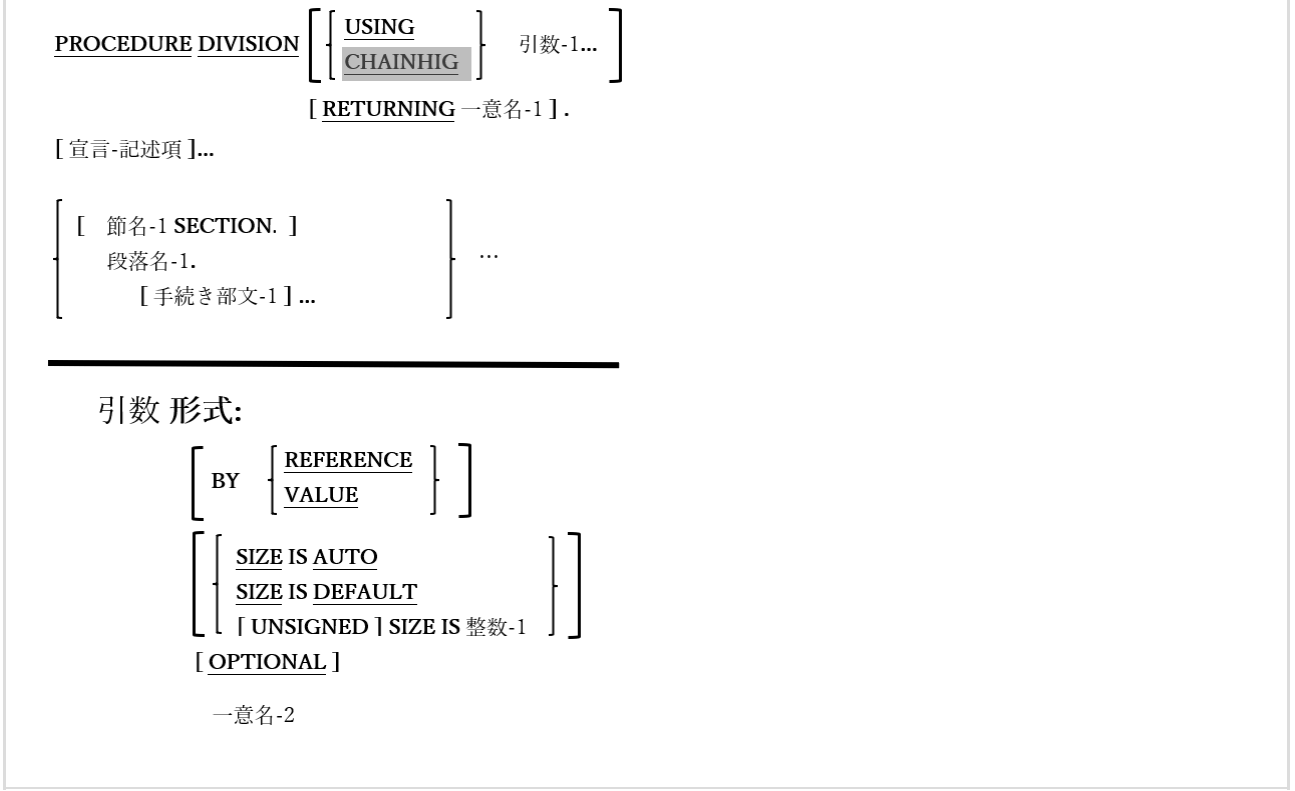
レコードロックオプション(すべてのオプションがすべての文で利用できるとは限らない)を次の表で示している。

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 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 LOCK WITH IGNORE 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。 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。 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。 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(8.2.4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ってはいるが、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、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)。

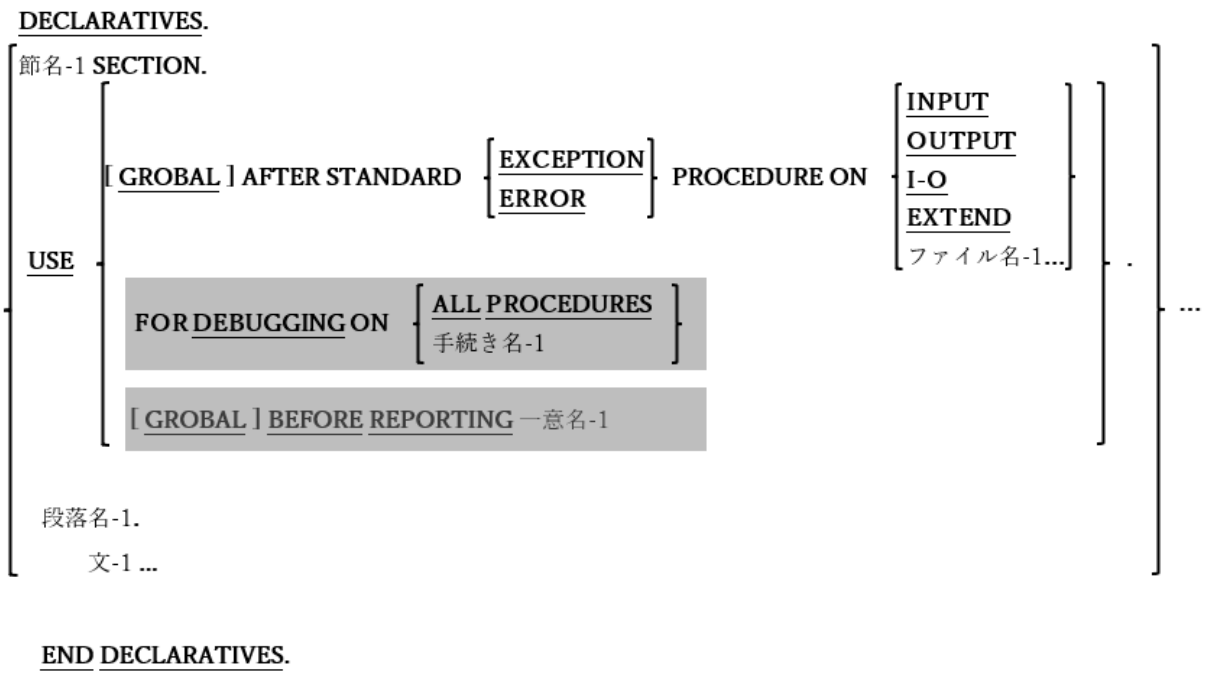
5. BY VALUE句では、引数に対応する呼び出し側プログラムからのデータ項目の読み込み専用コピーがプログラムに引き渡される。BY VALUE引数の内容は、サブプログラムによって変更することはできない。

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



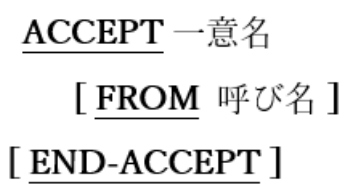
プログラマは手続き部の宣言領域内で、プログラム実行時に発生する可能性のある特定のイベントを遮断する、一連の「トラップ」ルーチンを定義することができる。

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)

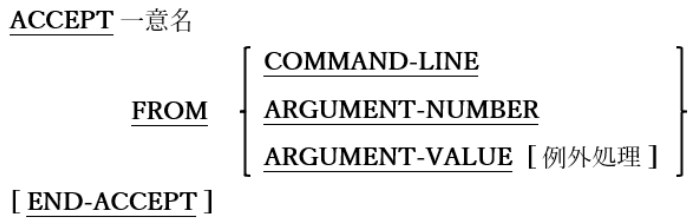


コンソールウィンドウから値を読み取り、それをデータ項目(一意名)に格納するために使用する。

1. FROM句を使う場合、指定する呼び名は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呼び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)



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。

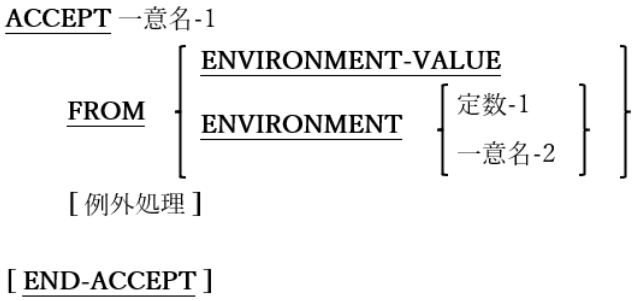
- Windowsシステムでは、一重引用符またはアポストロフィ文字(')は、他のデータ文字と同じように扱われ、文字列を示すことはできない。

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 `17` 。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使用法については、6.4.7で説明する。

`17` 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。

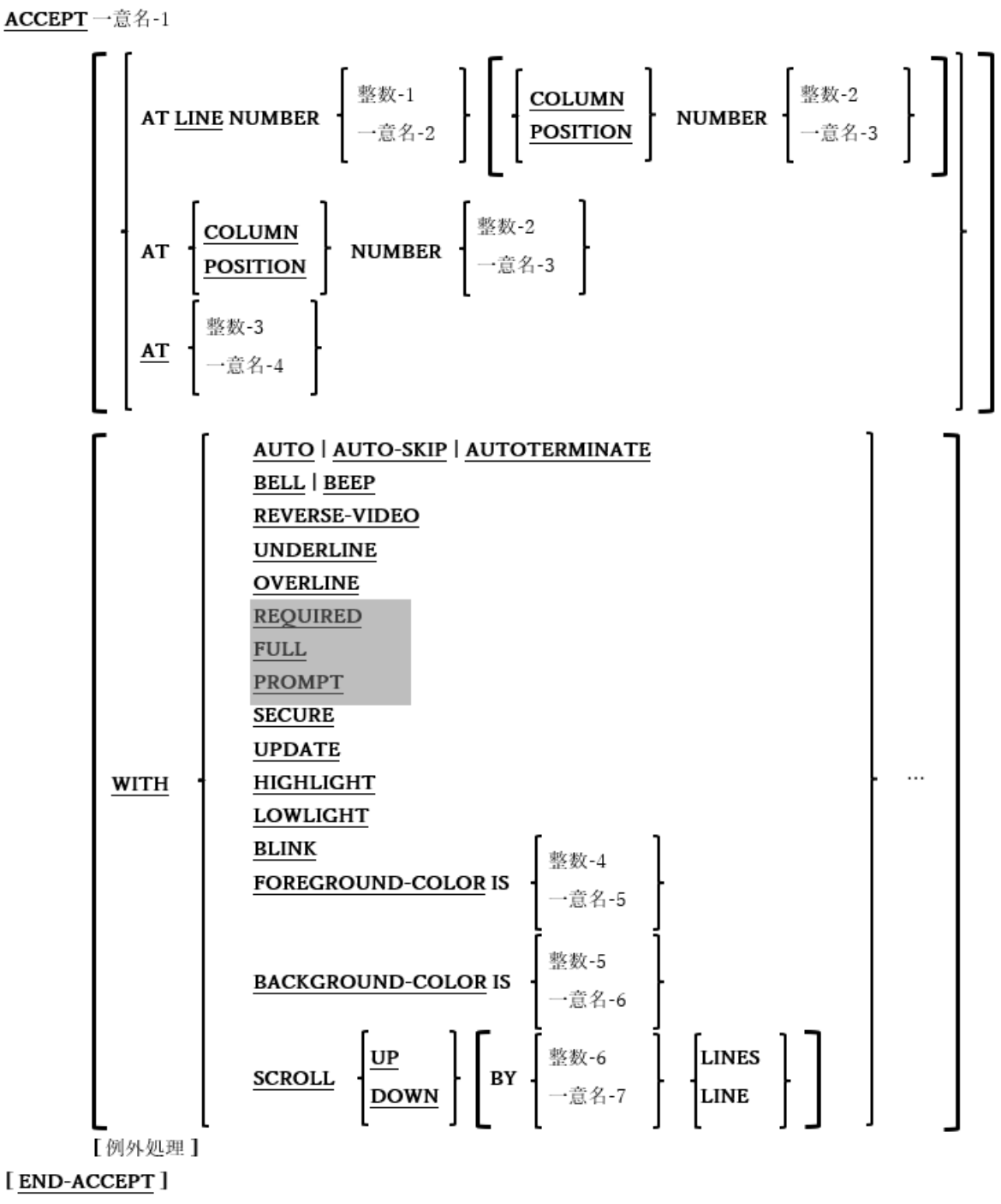
1. ENVIRONMENT-VALUEから受け取る場合、現在のENVIRONMENT-NAMEレジスタにある環境変数の値を取得するようにopensource COBOLランタイムシステムに要求する 18 。
2. 環境変数値を取得する、より簡単なアプローチは「ACCEPT ･･･ FROM ENVIRONMENT」を使うことである。その書き方では、ACCEPTコマンド自体で取得する環境変数を指定する。
3. オプションの例外処理の構文と使用法については、6.4.7で説明する。

---

18 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。

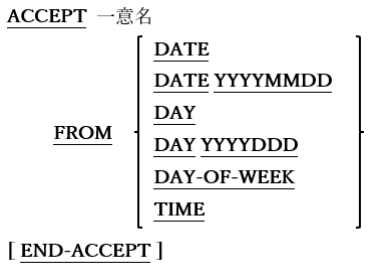
4. UPDATEオプションは、新しい値を受け取る前に一意名-1の現在の内容を表示する句である。

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。

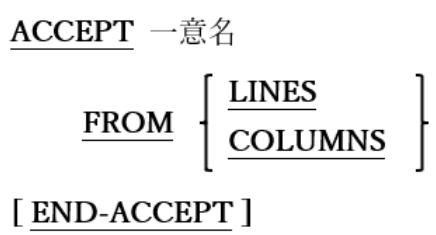
   表6-23-ACCEPTオプション(日付/時刻の取得)

オプション	取得データ	一意名-1の書き方
DATE	グレゴリオ暦表示の日付	01 CURRENT-DATE. 05 CD-YEAR PIC 9(2). 05 CD-MONTH PIC 9(2). 05 CD-DAY-OF-MONTH PIC 9(2).
DATE YYYYMMDD	グレゴリオ暦表示の日付	01 CURRENT-DATE. 05 CD-YEAR PIC 9(4). 05 CD-MONTH PIC 9(2). 05 CD-DAY-OF-MONTH PIC 9(2).
DAY	ユリウス暦表示の日付	01 CURRENT-DATE. 05 CD-YEAR PIC 9(2). 05 CD-DAY-OF-YEAR PIC 9(3).
DAY YYYYDDD	ユリウス暦表示の日付	01 CURRENT-DATE. 05 CD-YEAR PIC 9(4). 05 CD-DAY-OF-YEAR PIC 9(3).

DAY-OF-WEEK	曜日	01 CURRENT-DATE. 05 CD-DAY-OF-WEEK PIC 9(1). 88 MONDAY VALUE 1. 88 TUESDAY VALUE 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

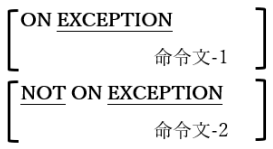


プログラムが実行されているコンソールウィンドウの(文字位置での)表示可能なサイズを取得するために使用する。

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

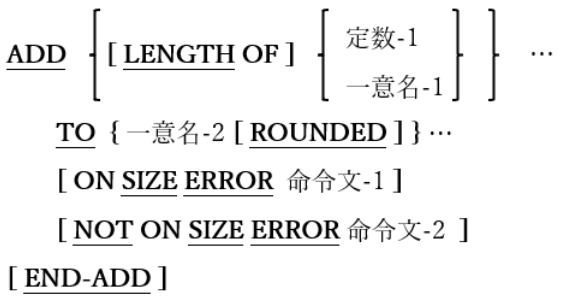


ACCEPT文の一部の書き方においてEXCEPTION句とNOTEXCEPTION句が利用可能で、ACCEPT文の失敗または成功時に実行されるコードを(それぞれ)指定できる。ACCEPT文ではリターンコードまたはステータスフラグを設定しないため、これが成功と失敗を検出する唯一の方法となる。

6.5. ADD

6.5.1. ADD文の書き方1 — ADD TO

図6-26-ADD TO構文



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は数字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. ONSIZE 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)。

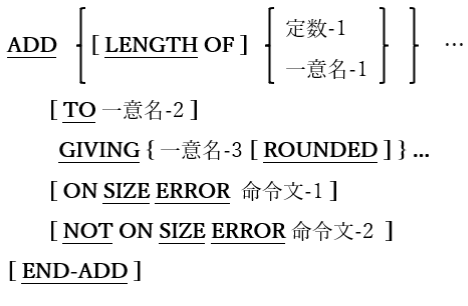
   図6-27-ON SIZE ERROR句を使用するサンプルプログラム

```
 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 GIVING構文



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 CORRESPONDING構文

```
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 ]
```

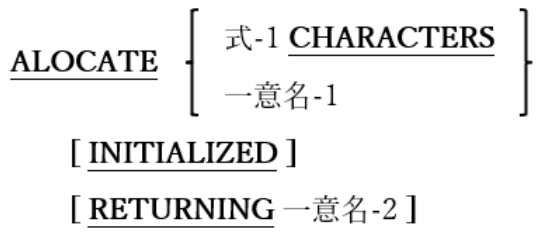
二つの一意名に従属して見つかったデータ項目に対応する個々のADD TO文と、同等のコードを生成する。

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。

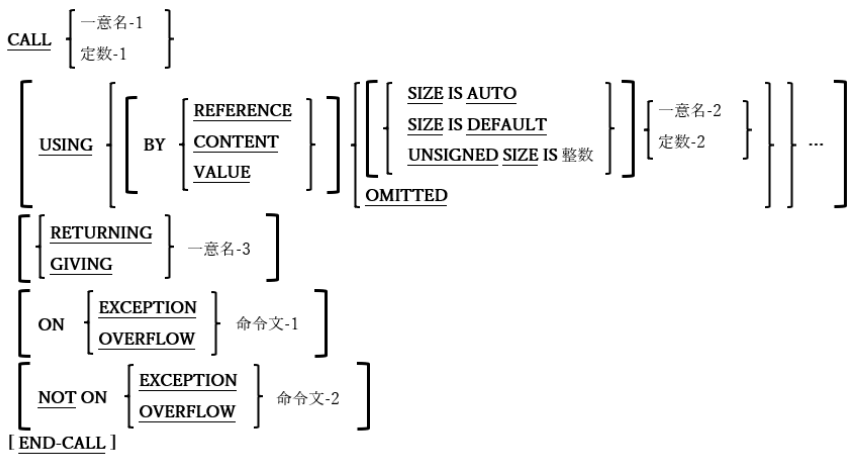
上記二つの使用法の唯一の機能上の違いとしては、最初の例で、INITIALIZED句がある場合は尊重されることである。

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 `19` 。

`19` 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indetifier-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方法の詳細については、8.1.4および8.1.5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、8.1.4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。

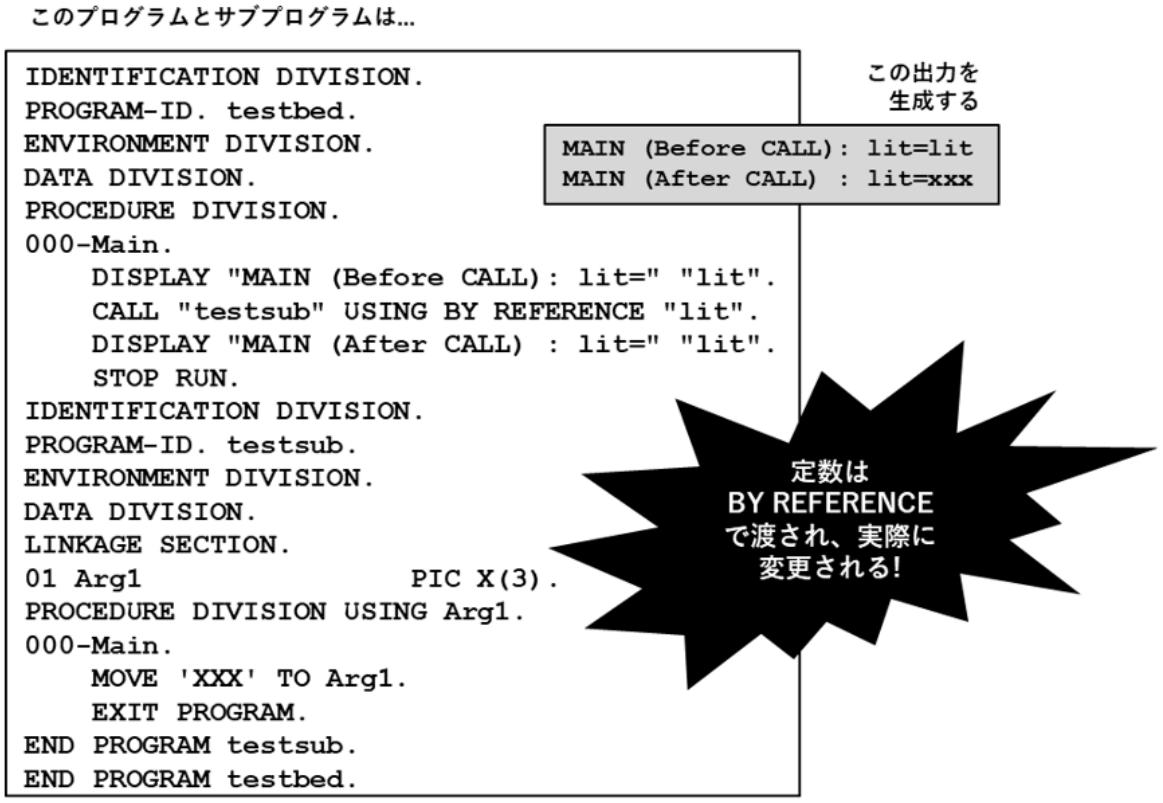
8. USING句では、CALLする側のプログラムからサブプログラムに渡される可能性のある引数のリストを定義する。引数が渡される方法は、BY句によって異なる。

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 `20` 。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。

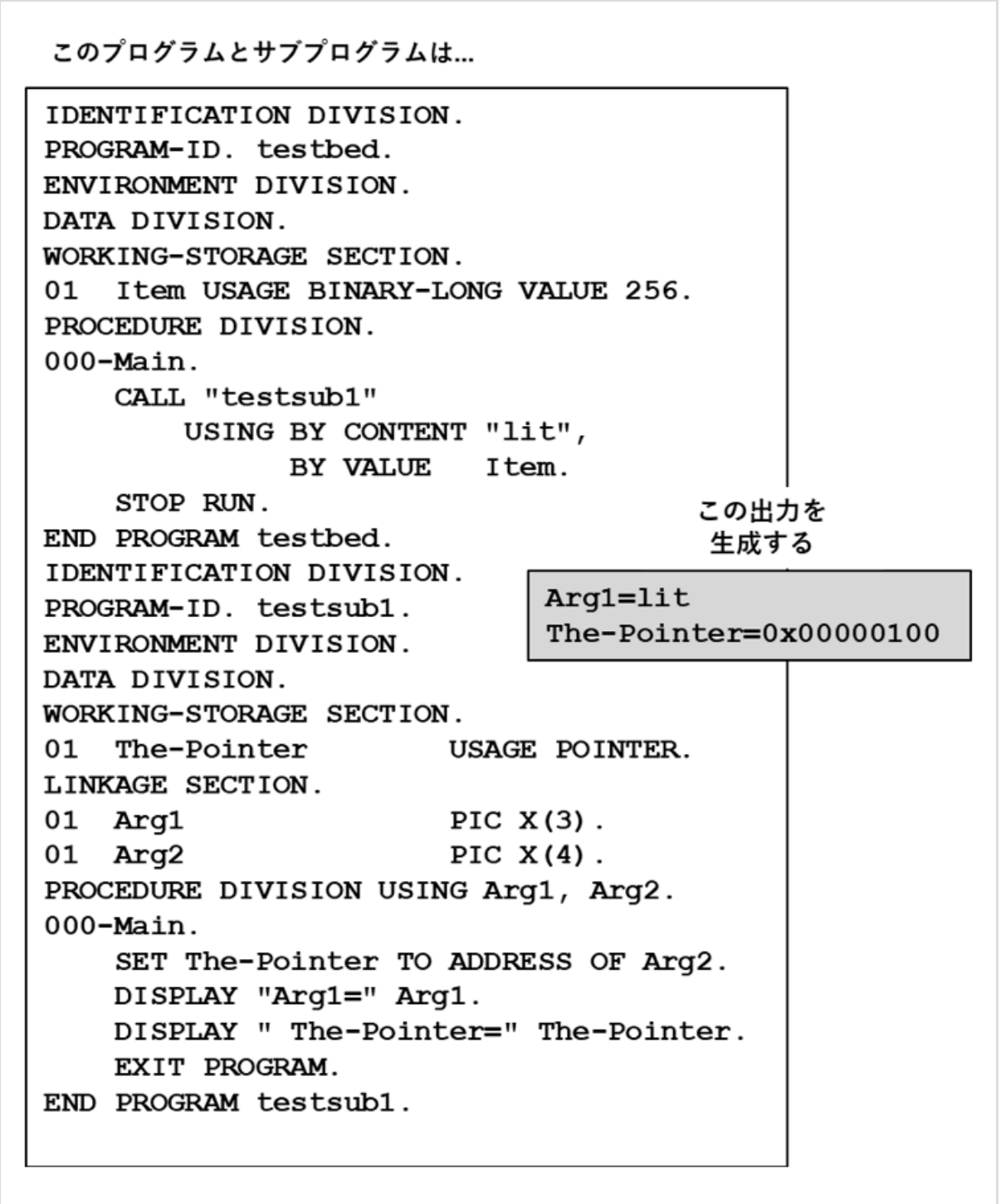
11. BY CONTENTは、引数のコピーのアドレスをサブプログラムに渡す。サブプログラムが引数の値を変更した場合、CALLする側のプログラムに戻された元のバージョンは変更されない。図6-32に示すように、これは定数値をサブプログラムに渡すための最も安全な方法である。

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。

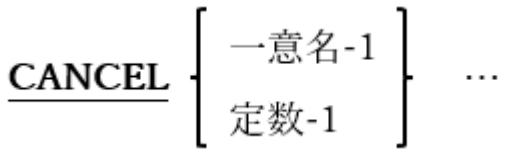
---

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

6.8. CANCEL

6.8.1. CANCEL文の書き方1 — CANCEL

図6-34-CANCEL構文



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。

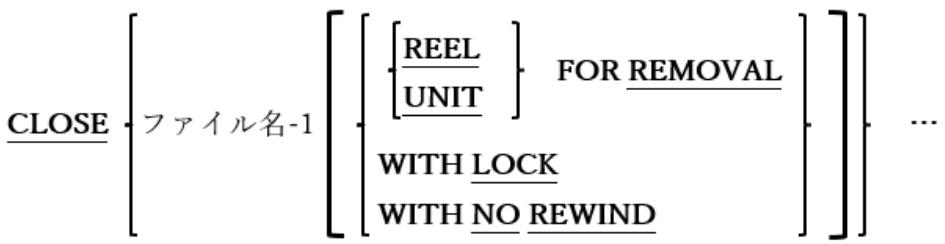
6.8.2. CANCEL文の書き方2 — CANCEL ALL

図6-35-CANCEL ALL構文

CANCEL ALL文は、一度でも呼ばれたすべての動的にロード可能なモジュールをメモリから破棄する。

6.9. CLOSE

図6-36-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ー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ー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

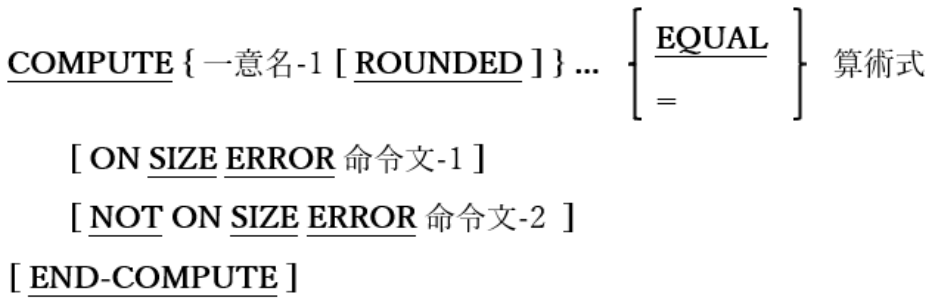
図6-37-COMMIT構文

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6-38-COMPUTE構文



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6-39-CONTINUE構文

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 (今後コードが必要になる可能性のある箇所も示す)
IF A = 1 IF B = 1 DISPLAY „A=1 & B=1" END-DISPLAY END-IF ELSE IF A = 2 IF B = 2 DISPLAY „A=2 & B=2" END-DISPLAY END-IF END-IF END-IF	IF A = 1 IF B = 1 DISPLAY „A=1 & B=1" END-DISPLAY ELSE CONTINUE END-IF ELSE IF A = 2 IF B = 2 DISPLAY „A=2 & B=2" END-DISPLAY ELSE CONTINUE END-IF ELSE CONTINUE END-IF END-IF

   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
IF Action-Flag NOT = 'I' AND 'U' DISPLAY 'Invalid Action-Flag' EXIT PARAGRAPH END-IF	IF Action-Flag = 'I' OR 'U' CONTINUE ELSE DISPLAY 'Invalid Action-Flag' EXIT PARAGRAPH END-IF

   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’
```

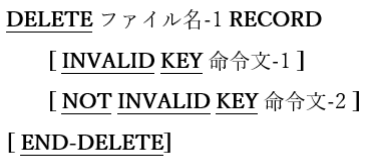
プログラマの多くは、「IF」を(誤って)「 `IF Action-Flag NOT = ‘I’ OR ‘U’` 」としてコーディングしていた。これにより、実行時に問題が発生することは避けられない。

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

6.13.1. DELETE文の書き方1 — DELETE

図6-40-DELETE構文(レコードの削除)



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。

6.13.2. DELETE文の書き方2 — DELETE FILE

図6-41-DELETE FILE構文(ファイルの削除)

```
DELETE FILE ファイル名-1
```

DELETE FILE文は、ディスクファイルを削除する。

1. ファイル名-1のファイルは、ディスクファイルでなければならず、DELETE FILE文を実行する前に閉じていなければならない。
2. ファイル名-1がVBISAMの場合は、拡張子が「.dat」と「.idx」のファイルを削除する。
3. DELETE FILE文の実行によって、ファイル名-1に関連するファイルステータス値が更新される。

6.14. DISPLAY

6.14.1. DISPLAY文の書き方1 — UPON CONSOLE

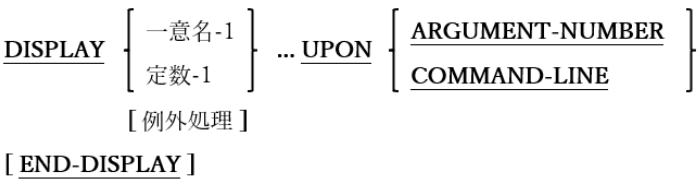
図6-42-DISPLAY構文(UPON CONSOLE)

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. 指定する呼び名は、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呼び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6-43-DISPLAY構文(コマンドライン引数へのアクセス)

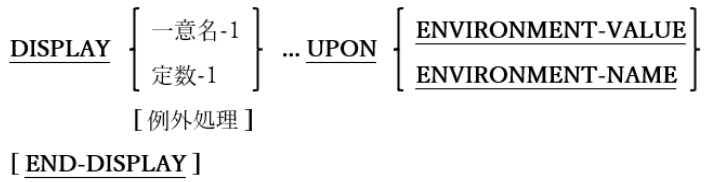


後続のACCEPTによって取得されるコマンドライン引数番号を指定したり、コマンドライン引数自体に新しい値を指定することができる。

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6-44-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。

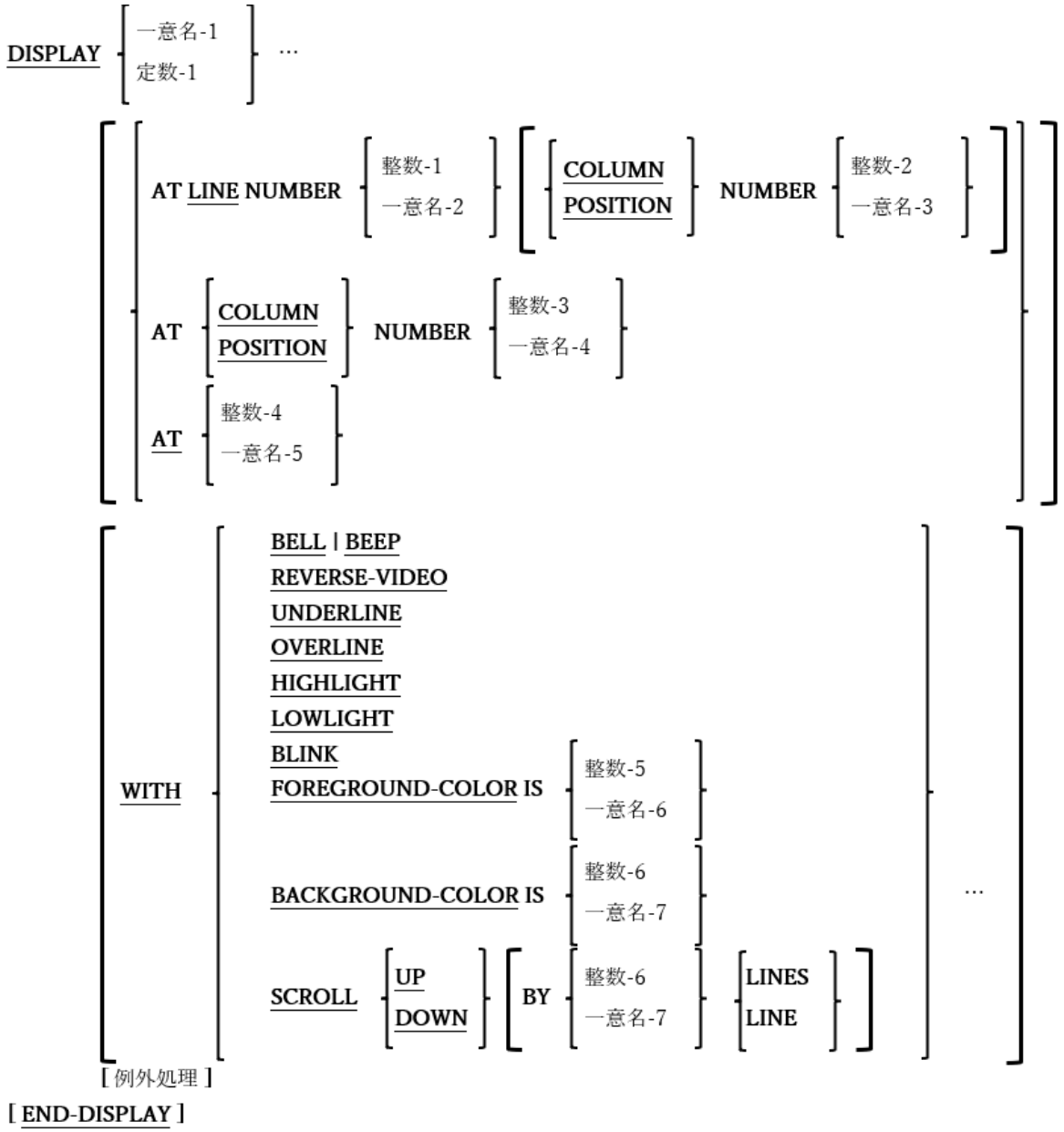
```
DISPLAY
    environment-variable-name UPON ENVIRONMENT-NAME
END-DISPLAY
DISPLAY
    environment-variable-name UPON ENVIRONMENT-VALUE
END-DISPLAY
```

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6-45-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。

1. 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。

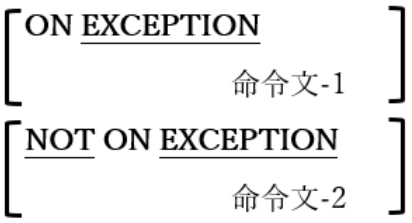
2. AT句では、データが画面に表示される前に、カーソルを画面上の特定の場所に配置することができる。定数-3/一意名-4の値は4桁である必要があり、最初の2桁はカーソルを配置する行、最後の2桁は列を示す。

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6-46-DISPLAY構文(例外処理)

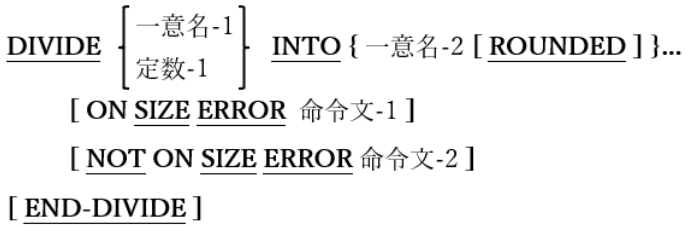


DISPLAY文のすべての書き方で使用可能なEXCEPTION句とNOT EXCEPTION句を使うことで、DISPLAY文の失敗、成功時のそれぞれに実行されるコードを指定することができる。DISPLAY文ではリターンコードやステータスフラグを設定しないため、これが成功と失敗を検出する唯一の方法となっている。

6.15. DIVIDE

6.15.1. DIVIDE文の書き方1 — DIVIDE INTO

図6-47-DIVIDE INTO構文

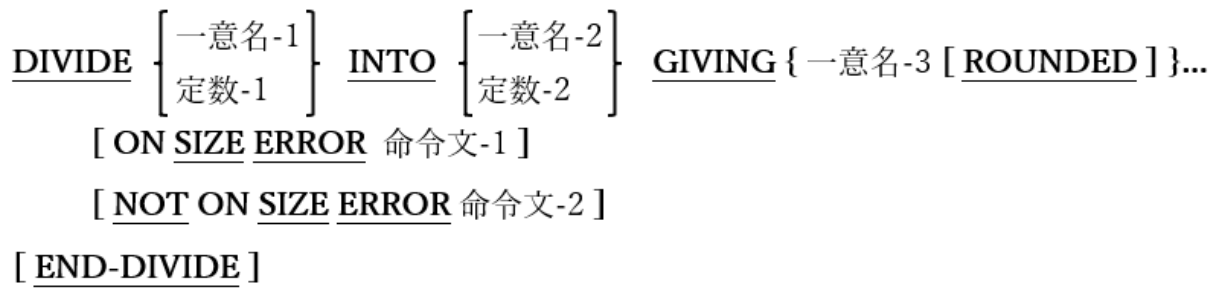


指定された値を一つ以上のデータ項目に分割し、それらの各データ項目を一意名-1または定数-1値で割った結果に置き換える。除算の余りは破棄される。

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6-48-DIVIDE INTO GIVING構文

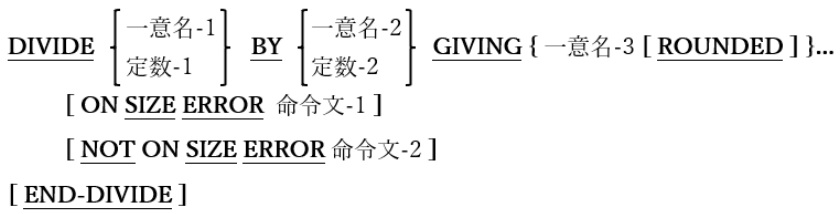


指定された値(一意名-1/定数-1)を別の値(一意名-2/定数-2)に分割し、一つ以上の受け取りデータ項目(一意名-3・・・)の内容を除算結果に置き換える。除算の余りは破棄される。

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6-49-DIVIDE BY GIVING構文



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6-50-DIVIDE INTO REMAINDER構文

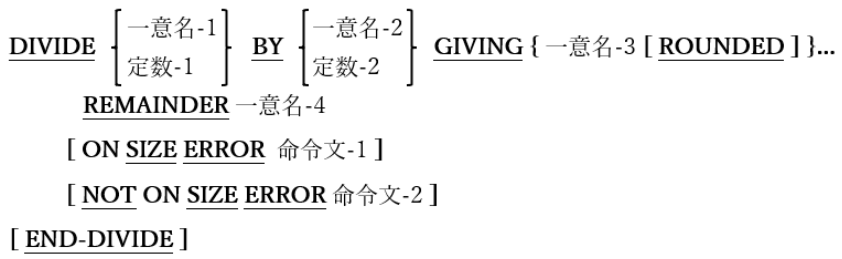
```
DIVIDE { 一意名-1 } INTO { 一意名-2 } GIVING { 一意名-3 [ ROUNDED ] }...
       { 定数-1   }      { 定数-2   }
    REMAINDER 一意名-4
  [ ON SIZE ERROR 命令文-1 ]
  [ NOT ON SIZE ERROR 命令文-2 ]
[ END-DIVIDE ]
```

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6-51-DIVIDE BY REMAINDER構文

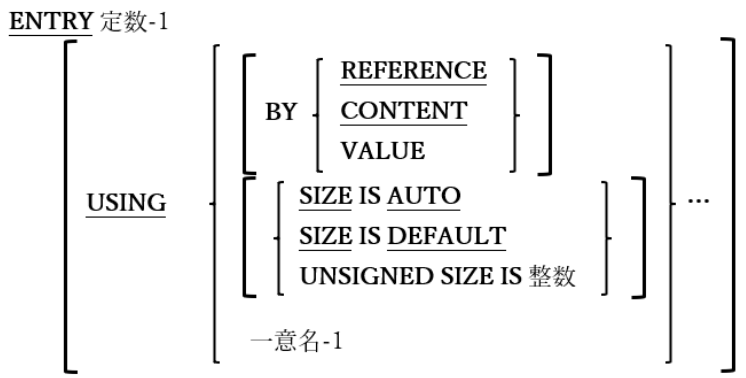


指定された値(一意名-1/定数-1)を別の値(一意名-2/定数-2)で除算し、一つの受け取りデータ項目(一意名-3・・・)の内容を除算結果に置き換える。除算の余りは一意名-4に格納される。

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6-52-ENTRY構文

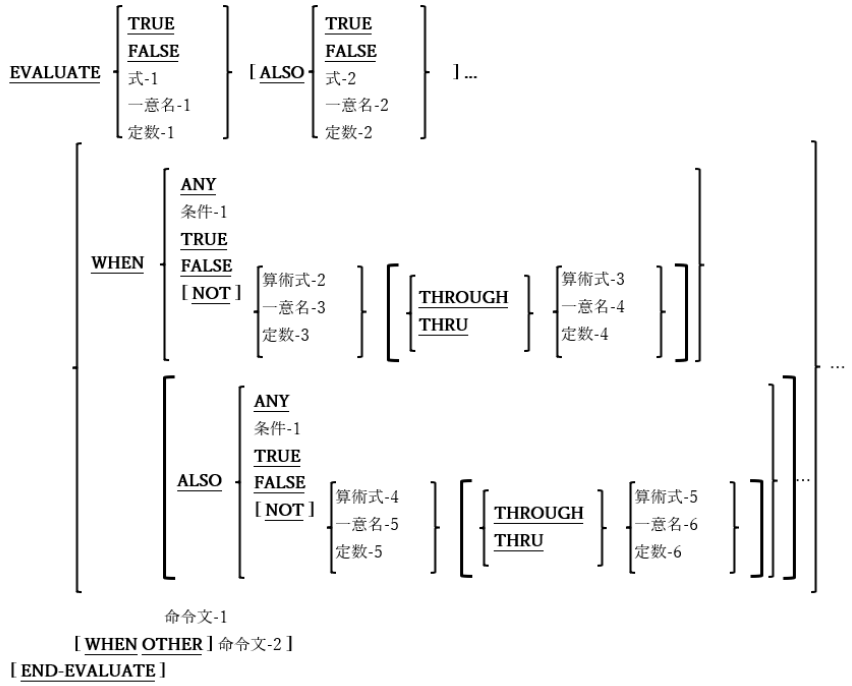


ENTRY文は、サブルーチンが予期する引数とともに、サブルーチンへの代替記述項ポイントを定義するために使用する。

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6-53-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:                              Not Legal:

(3 + Years-Of-Service) THROUGH 99      0 THRU “A”

“A” THRU “Z”                          Last-Name THRU Zip-Code(Assuming Last-Name is

X’00’ THRU X’1F’                       PIC X and Zip-Code is PIC 9)

15.7 THROUGH 19.4
```

3. EVALUATE文の後、最初のWHEN句の前に指定された値は選択主体と呼ばれ、各WHEN句の後に指定された値は選択対象と呼ばれる。
4. 各WHEN句には、EVALUATE文の選択主体と同じ数の選択対象が必要である。
5. 各EVALUATE句の選択主体は、選択対象に対応する各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の選択対象を使うと、ANYと一致する選択主体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。

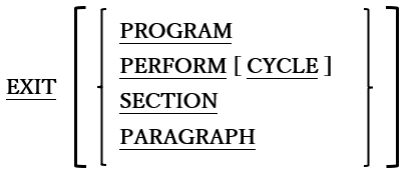
図6-54-EVALUATE文のデモプログラム

```
>>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6-55-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6-56-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
     GO TO 100-Process-A-Transaction-Exit.
   IF Input-Rec of Input-File = SPACES
     GO TO 100-Process-A-Transaction-Exit.  *>IGNORE BLANK RECORDS !
     process the record just read

100-Process-A-Transaction-Exit.
   EXIT.
```

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6-57は、EXIT PARAGRAPH文を使って、GO TOなしで図6-56の例をコーディングする方法を示している。

図6-57-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6-58は、図6-56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。

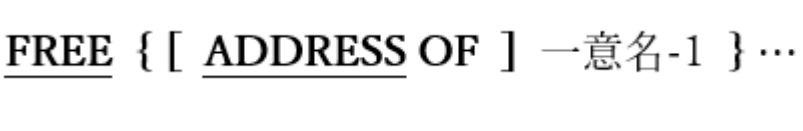
図6-58-EXIT PERFORM文

```
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6-59-FREE構文

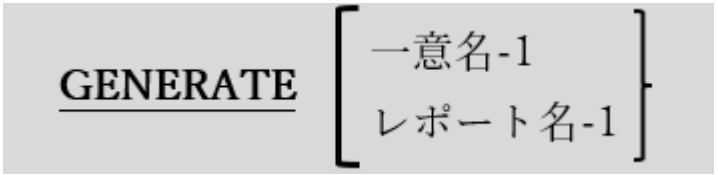


FREE文は、ALLOCATE文(6.6)によってプログラムに割り当てられていたメモリを解放する。

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6-60-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6-61-GOBACK構文

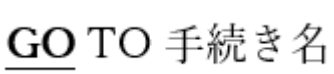
GOBACK文は、実行中のプログラムを論理的に終了するために使用する。

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6-62-GOTO構文



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6-63-GOTO DEPENDING ON構文

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えている場合、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。

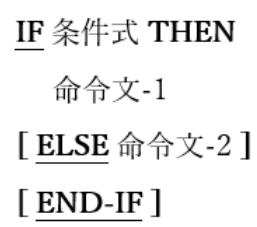
図6-64-GOTO DEPENDING ON vs IF vs EVALUATE

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

「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6-65-IF構文

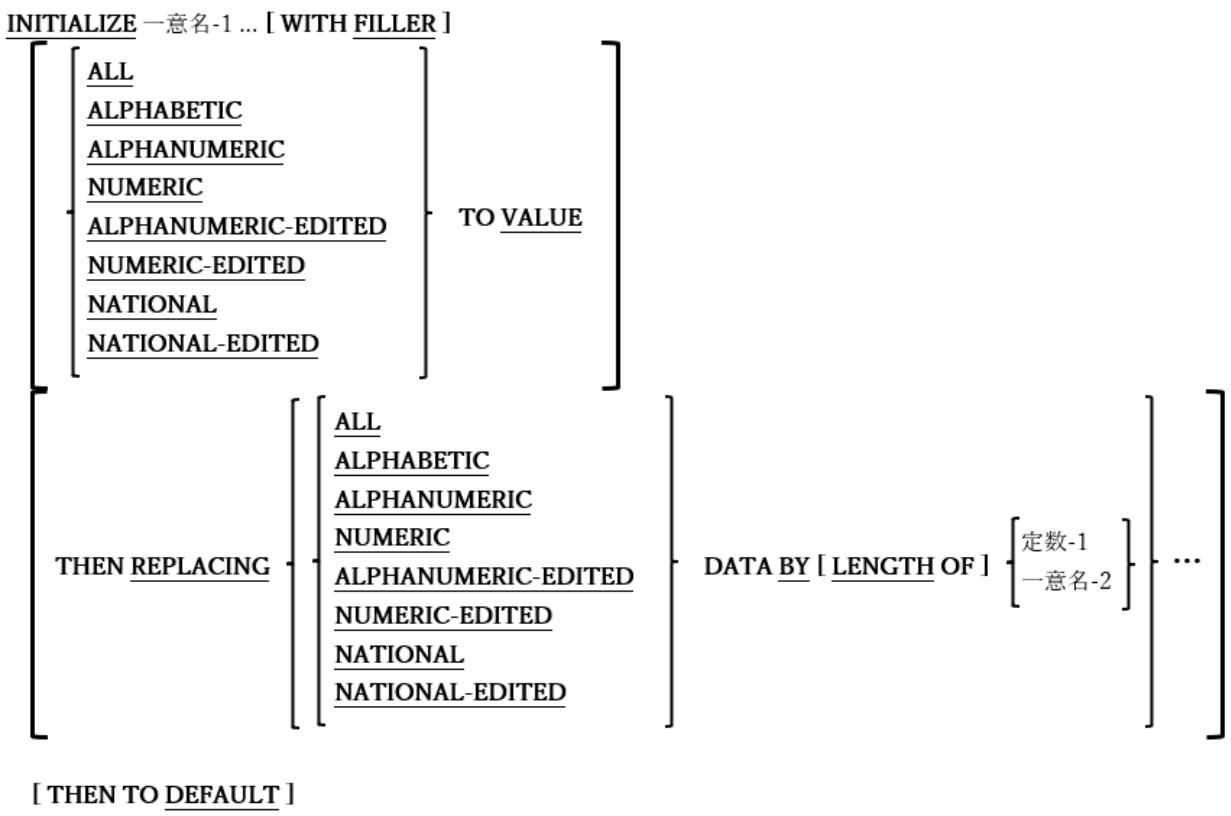


IF文は、一つの命令文を条件付きで実行するため、または条件式のTRUE/FALSE値に基づいて二つある命令文のうち一つを選択するために使われる。

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6-66-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。

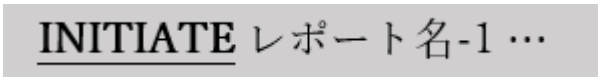
1. これによって新しい値に設定できるデータ項目のリストは次の通りである。
    - 一意名-1として指定されたすべての基本項目。
    - 一意名-1として指定され、集団項目に従属して定義されたすべての基本項目。以下の例外を除く：
        - USAGE INDEX項目は除外される。
        - 定義の一部としてREDEFINES句が含まれる項目は除外され、これに従属する項目も除外される。ただし、一意名-1の項目自体にREDEFINES句が含まれている場合や、REDEFINES句を含む項目に従属している場合がある。

    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

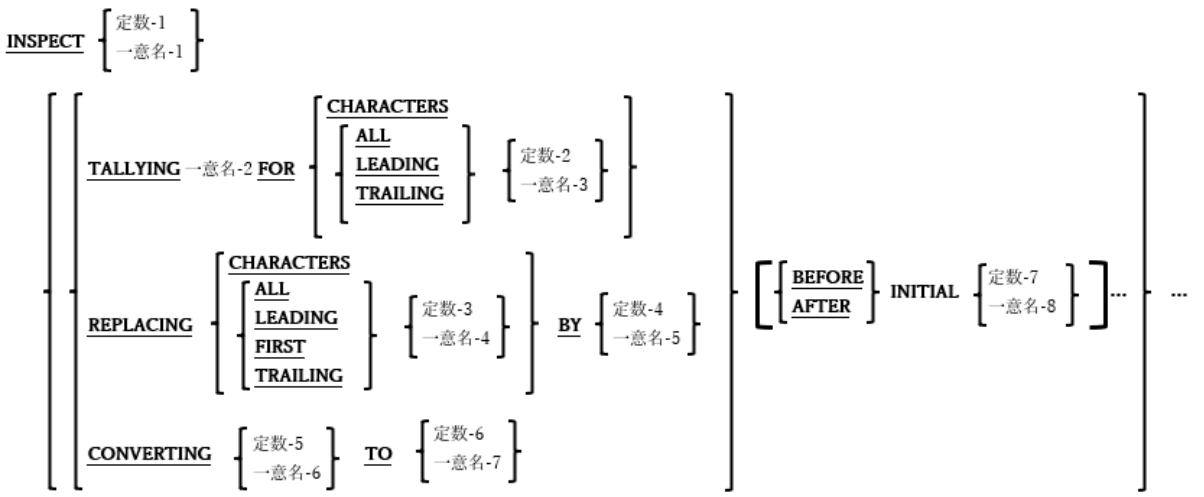
図6-67-INITIATE構文



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6-68-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。

1. 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がある。
2. 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。
3. 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。

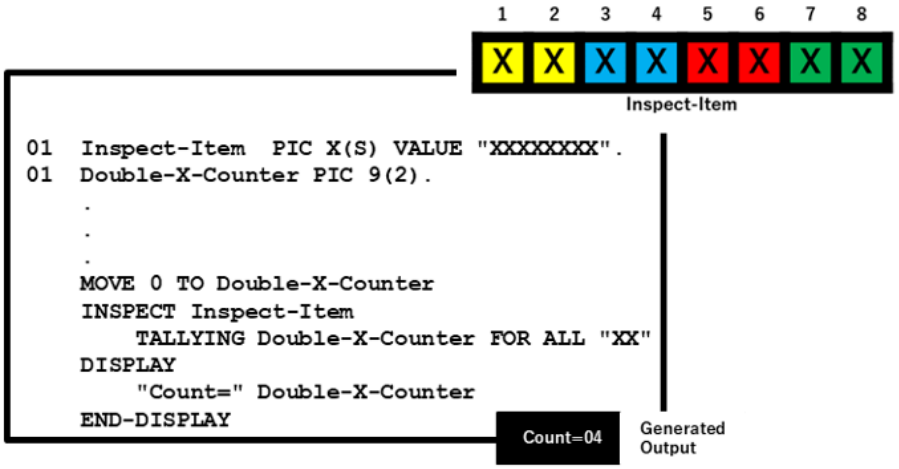
1. 一意名-2は編集不可の数値項目でなければならない。
2. 一意名-3および定数-2は、英数字のUSAGE DISPLAYデータとして明示的または暗黙的に定義する必要があり、この時一意名-3は集団項目の可能性がある。
3. 一意名-2は検索対象のターゲット文字列が一意名-1で見つかるたびに、1ずつ増加する。ターゲット文字列は以下のようになる。

   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6-69-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。

1. 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がある。
2. 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がある。
3. 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。
4. 「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。
5. 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。
6. 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。
7. 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。

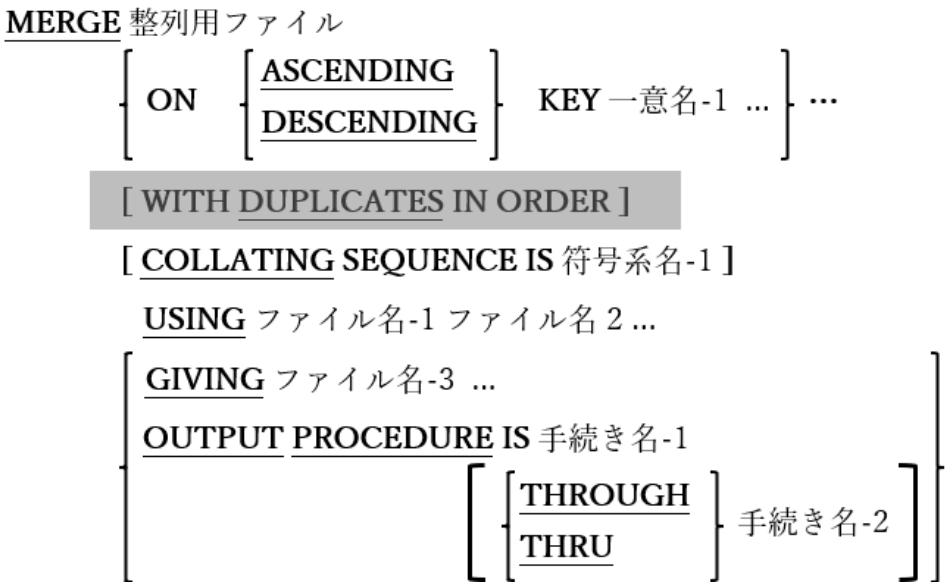
CONVERTING句の場合：

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り、この時一意名-5は集団項目の可能性がある。
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り、この時一意名-6は集団項目の可能性がある。
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6-70-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[21]。

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。

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。

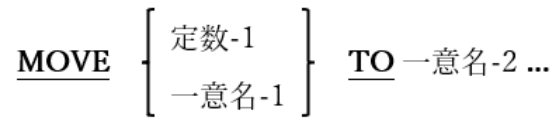
---

21 4.2.2参照。

6.28. MOVE

6.28.1. MOVE文の書き方1 — MOVE

図6-71-MOVE構文

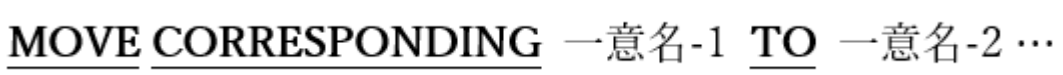


特定の値を一つ以上の受け取りデータ項目に移動することができる。

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6-72-MOVE CORRESPONDING構文



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は合致するごとに一つずつ、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----
```

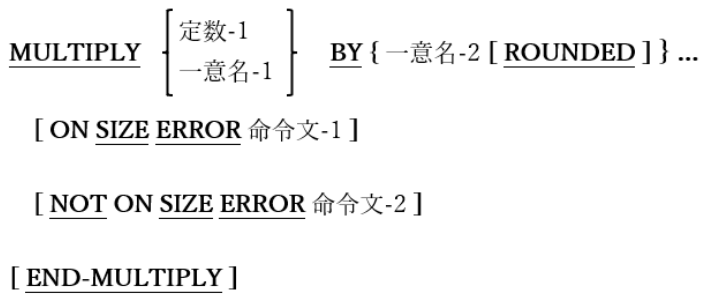
- opensource COBOLでは、「X」および「Y」集団項目内の「A」、「B」、および「C」データ項目間の「対応する」関係を確立している。「X」は 01-05-10-15 のレベル番号付けスキームを使用し、「Y」は 01-02-03-04 を使用しているが、この違いは対応するものの一致が確立することに影響しない。
- G OF X はOCCURS 句を含むデータ項目の親であるが、「G」項目が一致する。
- 「D」項目は3項のbに違反しているため、一致するものはない(4つの集団項目名を注視すること)。
- E OF X は3項のd(REDEFINES)に違反しているため、「E」項目と一致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

- F OF Y 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6-73-MULTIPLY BY構文

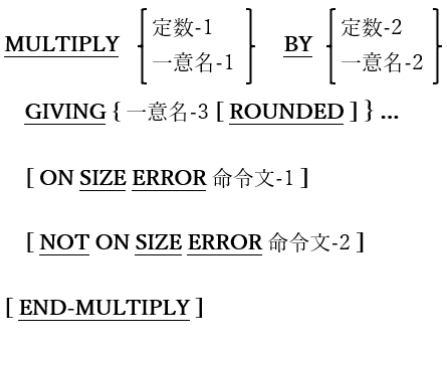


算術積を実行する。

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または定数-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6-74-MULTIPLY GIVING構文



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

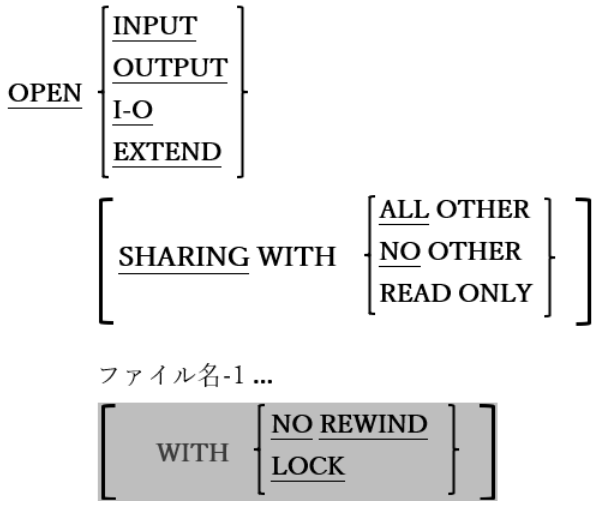
図6-75-NEXT SENTENCE構文

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6-76-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。

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6.1.8.1で説明している。

6. WITH NO REWIND句とWITH LOCK句は機能しない。

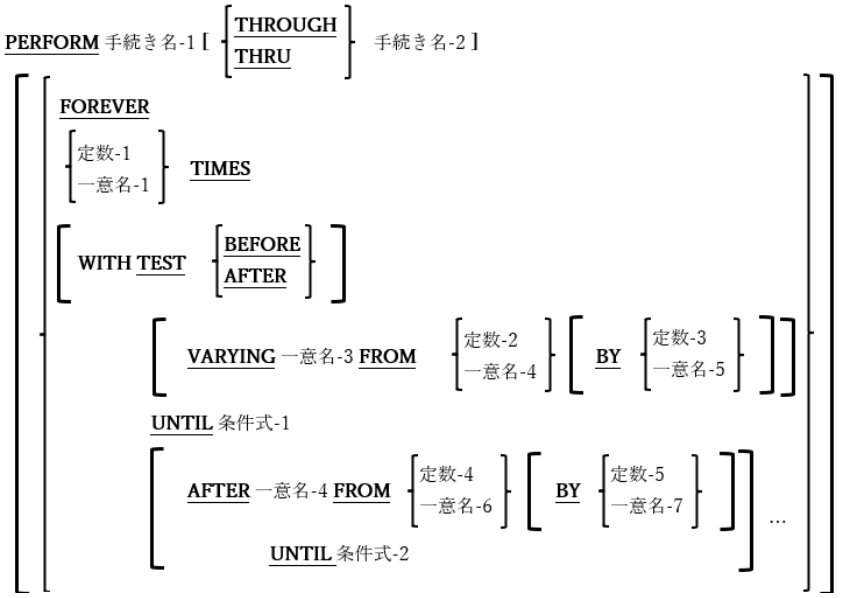
   WITH NO REWIND句をサポートできるデバイス(テープドライブ)は、opensource COBOLが動作する環境では非常に稀であり、コンパイラまたはランタイムメッセージは発行されない(何も実行されない)。

   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6-77-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。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.
```

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

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

```
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.
```

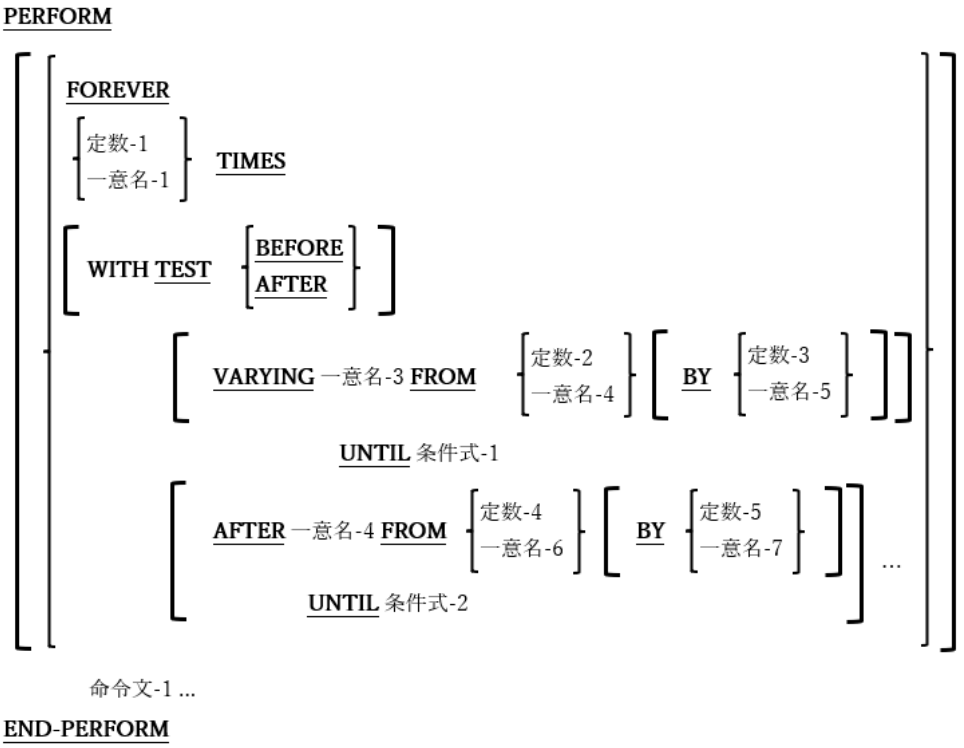
1	4	7	10
2	5	8	11
3	6	9	12

一方で上に示した順序で各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6-78-インライン型PERFORM構文



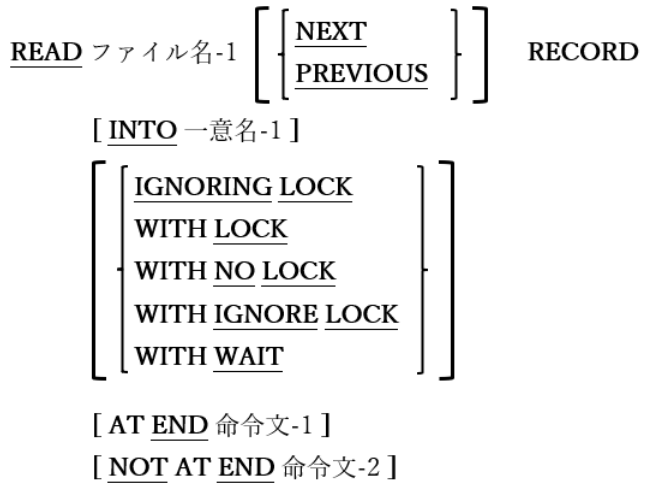
PERFORMの範囲内にある文が、プログラム内の他の場所にあるプロシージャではなく、PERFORMのコードにインラインで指定されること以外は、書き方1と同じである。

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6-79-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
   ```

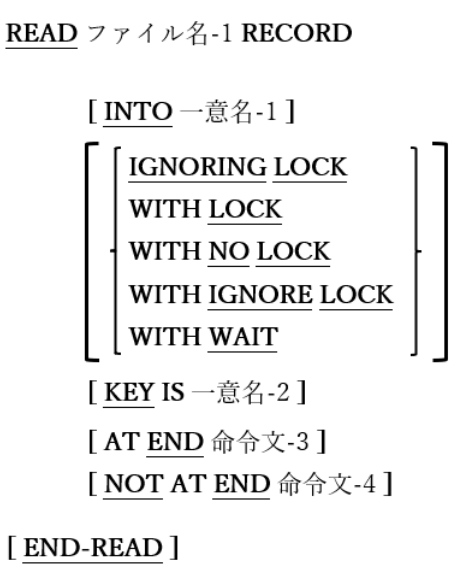
   ･･･正しい書き方として認められる。そのため、ACCESS MODE DYNAMICが指定されていて、上記のような文を順次読み取りとして処理することをopensource COBOLコンパイラに通知する場合は、文にNEXTまたはPRIORを追加する必要がある(そうでない場合は、ランダム読み取りとして扱われる)。
5. ファイル名-1で次に使用可能なレコードが取得され、その内容はファイルのFD(5.1)に従属する01レベルのレコード構造に格納される。
6. NEXT句およびPREVIOUS句では、読み取りプロセスがどの方向でファイルを通過するかを指定する。どちらも指定されていない場合は、NEXT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6.1.8.2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6-80-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：

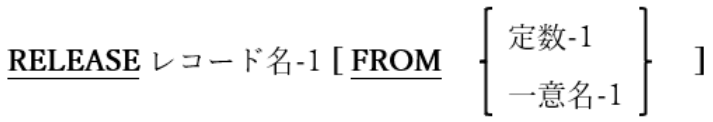
   - ファイルがORGANIZATION RELATIVEファイルの場合、一意名-2の内容が、アクセスされるレコードの相対レコード番号として使われる。一意名-2は、ファイルのRELATIVE KEY項目である必要はない(必要に応じて指定することが可能)。

- 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6.1.8.2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6-81-RELEASE構文

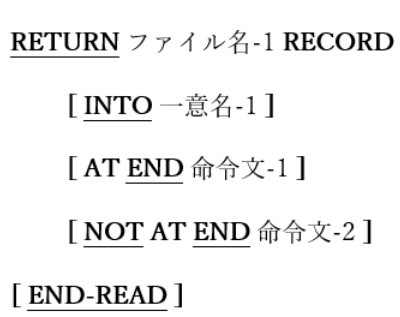


RELEASE文は、整列ファイルに新しいレコードを追加する。

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6-82-RETURN構文

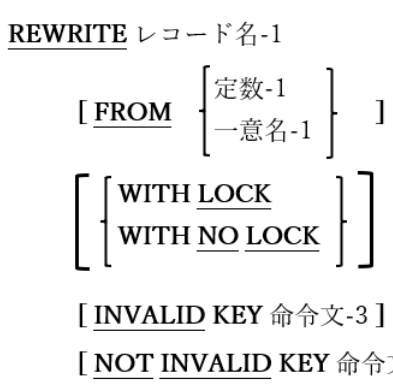


RETURN文は、整列ファイルまたはマージファイルからレコードを読み取る。

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6-83-REWRITE構文



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6.1.8.2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6-84-ROLLBACK構文

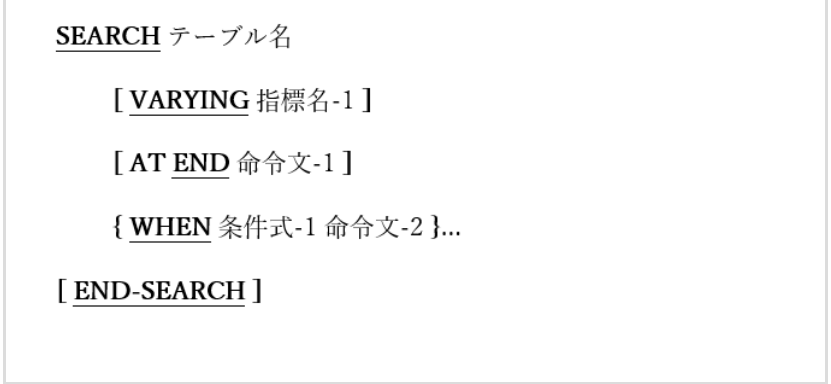
ROLLBACK文は、プログラムの開始以降または最後のCOMMIT以降に行われたすべてのファイルへの変更を元に戻す。

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6-85-SEARCH構文(順次探索)



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
```

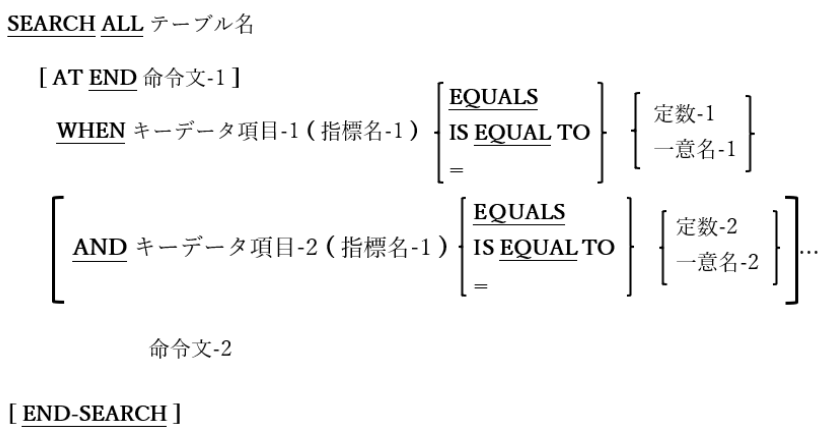
4. 探索プロセス中に条件式-1が評価され、TRUEの場合は命令文-2が実行された後に、制御はSEARCH文の次に移る。
5. 複数のWHEN句が存在する場合、それぞれの条件式-nが順番に評価され、最初にTRUEと評価された条件式に対応する命令文-nが実行された後に、制御はSEARCH文の次に移る。

6. TRUEと評価される条件式-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句の条件式-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. 条件式-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6-86-SEARCH構文(二分探索)



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。 22

   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。 23

   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log2n回の探索、記述項が存在しないことを示す時でもlog2n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(1,00010=11111010002)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。

---

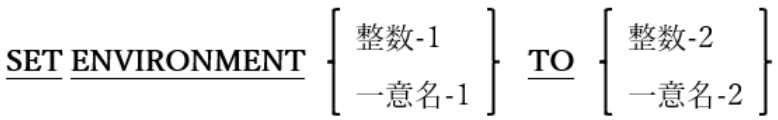
22 もちろん、データの順序がKEY句と一致しない場合は、テーブルソートを使って簡単に順序を揃えることができる(SORT文の書き方2–テーブルソートを参照)。

23 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)。

6.39. SET

6.39.1. SET文の書き方1 — 環境設定

図6-87-SET構文(環境設定)



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。

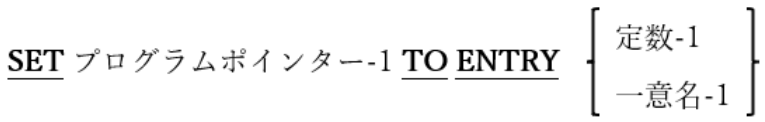
1. opensource COBOLプログラム内から生成または変更された環境変数は、そのプログラム(つまりCALL“SYSTEM”)によって生成されたすべてのサブシェルプロセスで使用できるが、opensource COBOLプログラムを開始したシェルまたはコンソールウィンドウには認識されない。

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。

```
DSIPLAY
"VALUE"
    “VARNAME” UPON ENVIRONMENT-NAME
END-DISPLAY
DSIPLAY
    “VALUE” UPON ENVIRONMENT-VALUE
END-DISPLAY
SET ENVIRONMENT “VARNAME” TO “VALUE”
```

6.39.2. SET文の書き方2 — プログラムポインター設定

図6-88-SET構文(プログラムポインター設定)



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。

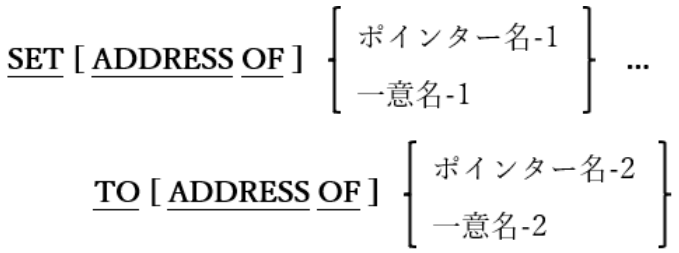
2. program-pointer-1はプログラムポインターとして使用しなければならない。

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、8.3.1.23および8.3.1.24で説明する。

6.39.3. SET文の書き方3 — アドレス設定

図6-89-SET構文(アドレス設定)

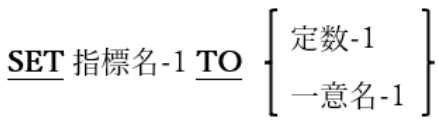


データ項目の内容ではなく、アドレスを処理するために使われる。

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6-90-SET構文(インデックス設定)

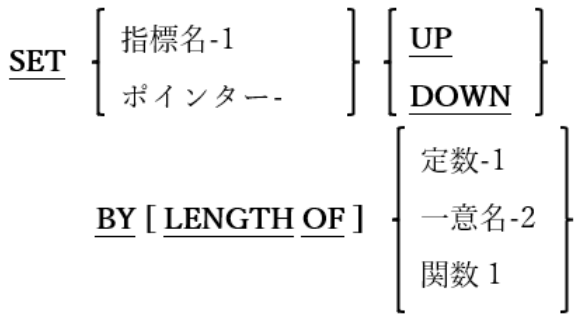


USAGE INDEXデータ項目に値を割り当てる。

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6-91-SET構文(UP/DOWN設定)

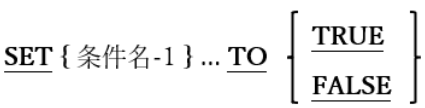


インデックスまたはポインタの値を指定された値の分だけインクリメントまたはデクリメントするために使われる。

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6-92-SET構文(条件名設定)

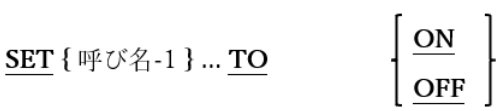


レベル88条件名のTRUE/FALSE値を指定することができる。

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6-93-SET構文(スイッチ設定)



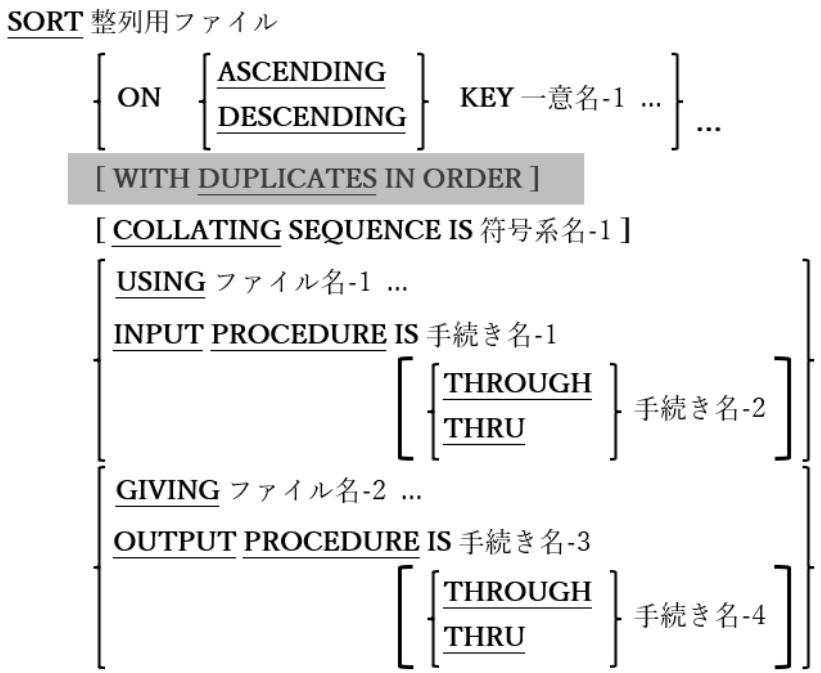
スイッチをオンまたはオフにする。

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6-94-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)を使用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(128MB) `24` 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。

```
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 `25` 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。

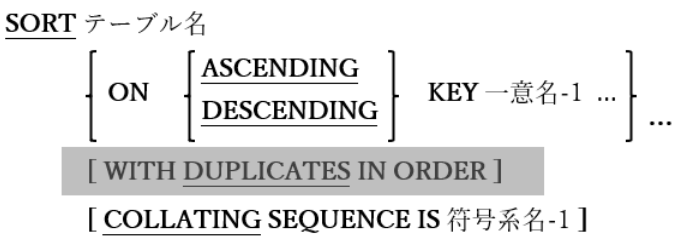
---

24 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)。

25 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。

6.40.2. SORT文の書き方2 — テーブルソート

図6-95-SORT構文(テーブルソート)

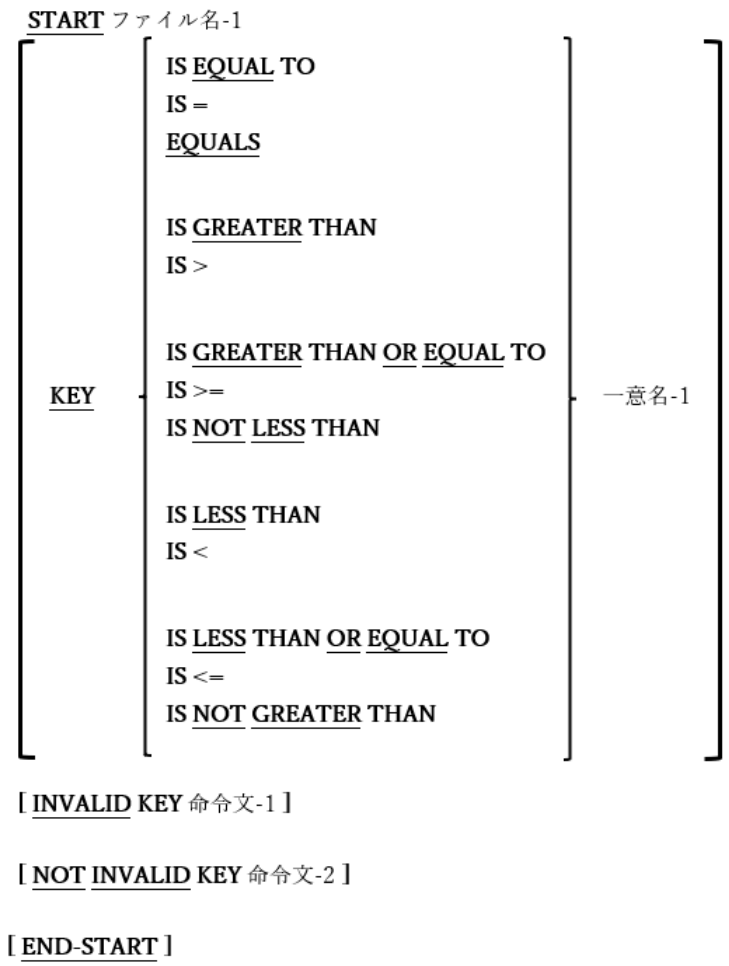


一つ以上のキー項目に従って、比較的少量のデータ、つまり、データ部のテーブルに含まれるデータを整列する。

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6-96-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。

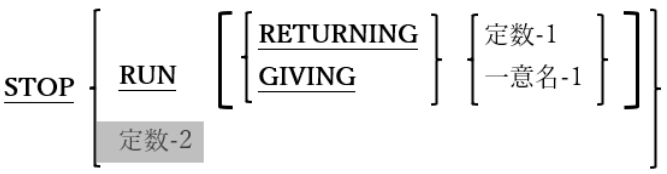
9. START文を実行中にエラーが発生した場合、ON INVALID KEY句がトリガーされる(つまり命令文-1が実行される)。このようなエラーは、入出力エラーまたは「キーが存在しない」エラー(ファイルステータス23)である可能性があり、KEY句の要件を満たすレコードが存在しないことを示す。

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6-97-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。

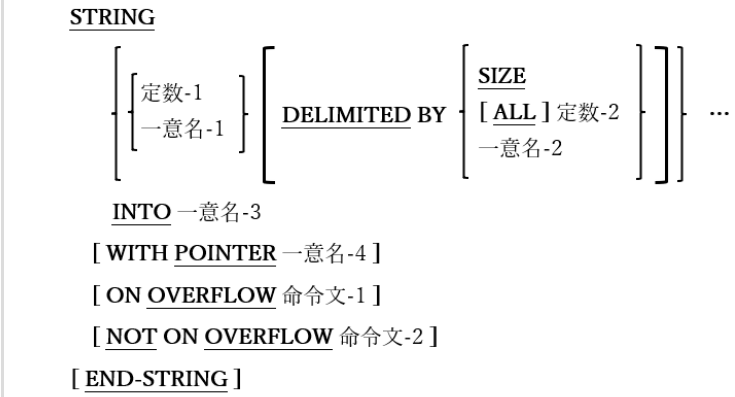
3. RETURNING句またはGIVING句を使うと、プログラムは数値リターンコードをオペレーティングシステムに返すことができ、リターンコードの値は、-2147483648から+2147483647の範囲にすることができる。

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6-98-STRING構文



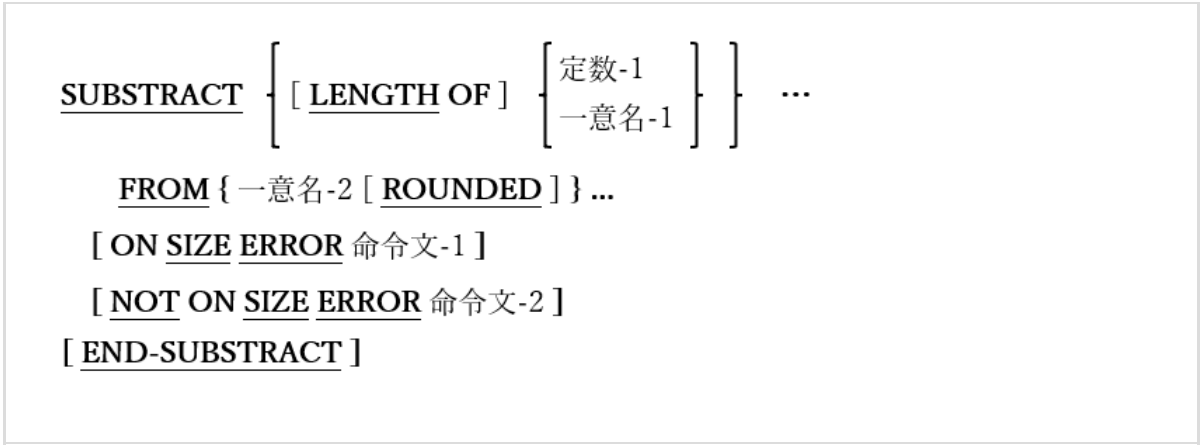
STRING文は、複数の文字列のすべて、または一部を連結して新しい文字列を形成するために使われる。

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ると、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も少ない場合に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6-99-SUBSTRACT構文

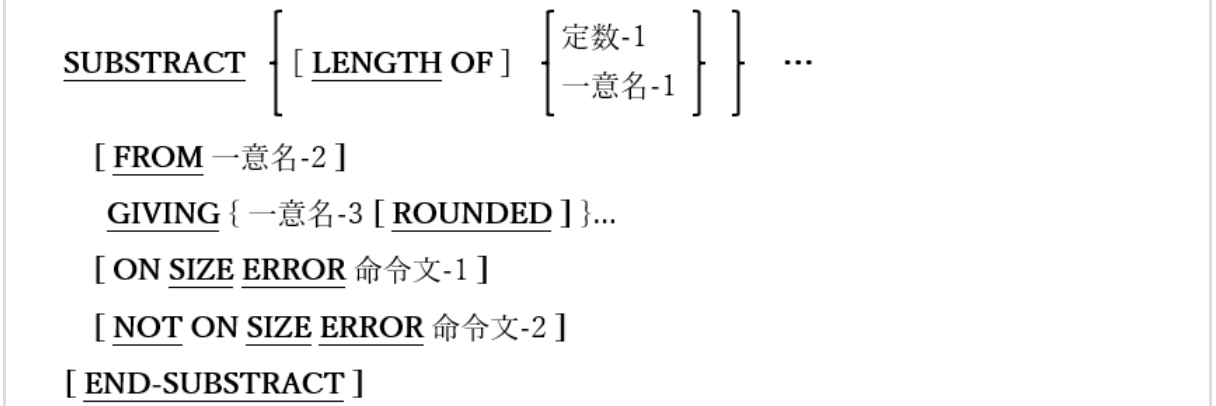


FROMの前にあるすべての引数(一意名-1または定数-1)の算術合計を生成し、その合計からTO(一意名-2)の後にリストされている各一意名を減算する。

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6-100-SUBSTRACT GIVING構文



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6-101-SUBSTRACT CORRESPONDING構文

```
SUBSTRACT 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-SUBSTRACT ]
```

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6-102-SUPPRESS構文

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。

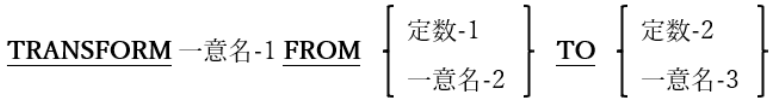
6.46. TERMINATE

図6-103-TERMINATE構文

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6-104-TRANSFORM構文



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6-105-機能的なTRANSFORM文

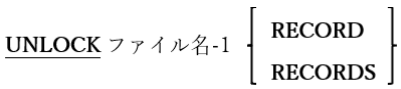
```
IDENTIFICATION DIVISION.
PROGRAM-ID. DEMOTRANSFORM.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Sample-Item PIC X(20) VALUE 'THIS IS A TEST'.
PROCEDURE DIVISION.
000-Main.
    TRANSFORM Sample-Item
         FROM 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
           TO 'ZYXWVUTSRQPONMLKJIHGFEDCBA'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

出力結果

```
GSRH RH Z GVHG
```

6.48. UNLOCK

図6-106-UNLOCK構文



この文は、まだ書き込まれていないファイルI/Oバッファー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。

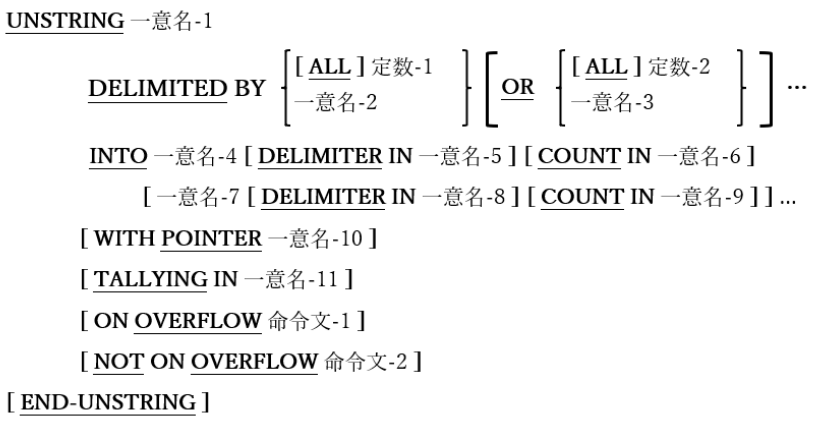
1. ファイル名-1がSORTファイルの場合、アクションは実行されない。
2. すべてのopensource COBOL実装がロックをサポートしているわけではない。それらが構築されたオペレーティングシステムと、opensource COBOLが生成されたときに使用されたビルドオプションによって異なる。 26 これらのopensource COBOL実装の一つを使用するプログラムがUNLOCKを発行すると、プログラムは無視されてコンパイラメッセージは発行されない。必要に応じて、バッファー同期は引き続き行われる。

---

26 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6-107-UNSTRING構文

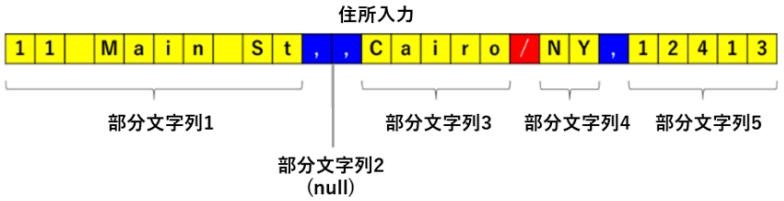


UNSTRING文は文字列を解析し、そこから部分文字列を抽出する。

1. 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。
2. 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。
3. 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。
4. 一意名-10の値は0より大きい必要がある。
5. 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6-108-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文が実行されたかのようになる。

```
MOVE   "11 Main St”      TO  Street-Address
MOVE   "”                TO  Apt-Number[27]
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
MOVE   ",”          TO  D4
MOVE   SPACES       TO  D5[28]
```

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では行われないため、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。

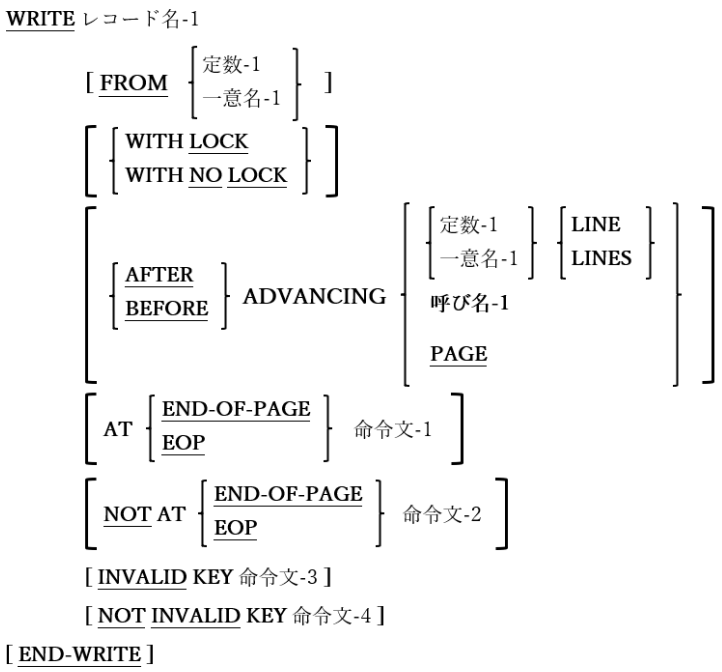
---

27 空白文字列のMOVEは、空白のMOVEと同じである。

28 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6-109-WRITE構文



WRITE文は、OPENファイルに新しいレコードを書き込む。

1. 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。
2. 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6.1.8.2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 ADVENCING 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. 日本語の使用

日本におけるコード系の標準は、JIS X0201のローマ文字・カタカナ用8単位符号系である。opensource COBOLでは、シフトJISコードはこのコード系に基づいて日本語文字のマッピングを行っている。

7.1. 英数字項目の日本語

文法上、日本語項目はPICTURE句の文字「Ｎ」でしか定義できないが、英数字項目でも日本語データ(文字と日本語文字の混在または日本語文字のみ)を取り扱えられるようにしてある。これは、文法上何も規定せず(整合性がとれなくなる)に、その使用はプログラマの責任としている。つまり、INSPECT文、STRING文およびUNSTRING文で使用した場合や、部分参照を行った場合、その実行結果は保証されない。このようなことを暗に認めているのは、PIC Nで定義した項目は日本語文字だけしか定義、格納できないが、実際のアプリケーション上では、文字(1バイトコード＝半角文字)と日本語文字(2バイトコード＝全角文字)が混在したデータが多数存在することによる。また、特に文法拡張を行わずに、PIC Xで日本語データを処理している既存製品との互換性をとる意味もある。例えば以下のように、日本語1文字に対して、2バイトの領域を定義する必要がある。

```
        01 データ項目1 PIC X(8) VALUE"顧客code"

        01 データ項目2 PIC X(10) VALUE"顧客コード"
```

データ項目1のように半角文字と全角文字が混在していると、プログラムの可搬性のために、コンパイル中に警告メッセージが表示されるが、実行は正常になされる。

注意：暗黙事項として、英数字項目でも日本語文字を格納できることとしているにも関わらず、日本語項目を新たに定義しているのは、次の2点が理由である。

1. NATIONAL(日本語)文字に対する処理系の標準化動向(日本語データの文字列操作を容易に行うこと)。
2. 種々の日本語コード系に対応を図る(シフトイン / アウト制御コードの削除)

7.2. 日本語項目と表意定数

日本語項目(PIC N項目)における各表意定数の値は、次の通りである。

表7-1-日本語項目と表意定数の値

表意定数	シフトJISコード
SPACE(S)	日本語空白文字 X"8140"
HIGH-VALUE(S)	X"FFFF"
LOW-VALUE(S)	X"0000"
ALL 定数	定数の値に依存する

7.3. 各命令文と日本語の取扱い

7.3.1. MOVE文

MOVE文で、英字、英数字、整数、英数字編集及び数字編集項目と日本語項目との転記を認めている。このことは、INSPECT文、STRING文及びUNSTRING文で、日本語文字(全角文字)と文字(半角文字)との混在を禁止しているので、文法上の整合はとれないが、PIC X項目による日本語の定義と格納と同様、実アプリケーション上の必要性があるということで転記を認めている。

表7-2は、転記時の処理内容を示すものであるが、送出し側データ項目には、文法上規定された正しいデータが格納されているものとする。

表7-2-転記の処理方法

送出し側データ項目の項類	受取側データ項目の項類	処理方式
英字	日本語，日本語編集	全角文字へコード変換
英数字	日本語，日本語編集	全角文字へコード変換
英数字編集	日本語，日本語編集	全角文字へコード変換
整数	日本語，日本語編集	全角文字へコード変換
非整数	日本語，日本語編集	コンパイルエラー
数字編集	日本語，日本語編集	全角文字へコード変換
日本語，日本語編集	英字	そのまま転記
日本語，日本語編集	英数字，英数字編集	そのまま転記
日本語，日本語編集	整数，非整数，数字編集	コンパイルエラー
日本語，日本語編集	日本語，日本語編集	そのまま転記

データの内容は、文字データのみ、日本語文字データのみ、および文字と日本語データが混在している場合がある。

文字には半角カタカナも含まれる。

ここで、文法上定義されていないのは、英字、英数字、英数字編集データ項目の内容が、日本語文字のみ、または文字と日本語文字が混在している時の処理方法である。この場合、送出し側データ項目の内容がすべて日本語文字(全角文字)の場合は、そのまま転記する。日本語文字(全角文字)と文字(半角文字)が混在しているときは、文字は全角文字へ変換を行い、日本語文字はそのまま転記する。なお、集団項目は英数字項目の扱いになるため、受取り側の各基本項目が日本語項目であっても、全角文字へのコード変換は行われない。転記は、標準桁寄せ規則に従って、必要に応じて右端を切り捨てたり、日本語空白文字の空白詰めを行う。ただし、送出し側が日本語データ項目で、受取り側データ項目の英字、英数字、英数字編集項目が2バイト単位のデータを格納できない(最後の1バイト領域へ全角文字を転記)場合には、最右端の最後のバイトは空白文字に置き換えられる。受取り側データ項目にJUSTIFIED句（けたよせ）句を書いた場合、桁寄せは、5.3に示すJUSTIFIED RIGHT句の規則に従う。

受取り側データ項目が日本語、日本語編集のとき、送出し側データ項目の内容によっては、次のように転記される。

表7-3-送出し側データ項目の内容に対する処理方法

送出し側データ項目の内容	処理方式
正しい文字	全角文字へコード変換

不正な文字(≠日本語文字)	日本語空白文字へコード変換
正しい日本語文字	そのまま転記
不正な日本語文字(≠文字)	そのまま転記
X”00”	X”0000”[29]
X”20” = 半角の空白文字	日本語空白文字へコード変換
X”FF”	X”FFFF”
制御コード，グラフィック文字	日本語空白文字へコード変換

ただし、日本語空白文字は、シフトJISコード系ではX”8140”である。

---

29 opensource COBOL 1.5.2Jではそのまま転記される不具合が発生している。

7.3.2. ACCEPT/DISPLAY文

ACCEPT文とDIPLAY文による日本語データの入出力も、実質的には、PICTURE句(5.3)および本章の英数字項目の日本語(7.1)とMOVE文(7.3.1)の規則に従って処理される。

日本語項目への入力では、日本語文字(全角文字)だけを受け取る。このとき、キーボード上の文字(JIS X0201 8単位符号)は、そのまま入力すると自動的に全角文字へ内部表現形式の変換を行う。また、必要に応じて、日本語空白文字を埋める。英数字項目に対しては、英数字文字(カタカナを含む半角文字)と日本語文字(全角文字)の入力が可能で、それらが混在していてもよい。ただし、受取り側データ項目が全角文字の入力に対してそのデータを格納できない(最後の1バイトの領域)場合には、最右端の文字位置は空白文字に置き換えて再表示される。いずれにしても文法上の規定外にあるため、その後の処理については注意が必要である。

7.4. UTF-8の使用

opensource COBOLは、Unicode(UTF-8)をサポートしている。この文字コードを使用する場合には「./configure」実行時に「--enable-utf8」を指定してビルドする必要がある。指定しない場合は、既定値のSHIFT-JISとなる。SHIFT-JISサポート版との違いは以下である。

1. PICTURE句において、「N」1つは3バイトと見なす。
2. 部分参照の開始位置と長さやINSPECTの単位は、「文字」ではなく「バイト」である。
3. 空白詰めは半角空白で行われる。
4. STRING文において、項目の種類が混在した時のチェックを抑止する。

8. opensource COBOLシステムインターフェース

8.1. opensource COBOLコンパイラの使い方(cobc)

8.1.1. 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。

8.1.2. コンパイルオプション

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。

```
使い方: cobc [options] file...
オプション:
  --help               このメッセージを表示します
  --version, -V        コンパイラのバージョンを表示します
  -v                   コンパイラが起動したプログラムを表示します
  -x                   実行可能プログラムをビルドします
  -m                   動的ロード可能モジュールをビルドします(デフォルト)
  -std=<方言>          指定した方言に基づいて警告/機能します :
                         cobol2002   Cobol 2002
                         cobol85     Cobol 85
                         ibm         IBM互換
                         mvs         MVS互換
                         bs2000      BS2000互換
                         mf          Micro Focus互換
                         default     指定しない
                         config/default.conf および config/*.conf を参照してください
  -free                  自由形式を使用します
  -free_1col_aster       自由形式(かつ第1カラムの*を注釈行の標識とみなす)を使用します
  -fixed                 固定形式を使用します(デフォルト)
  -O, -O2, -Os           最適化を有功にします
  -g                     Cコンパイラのデバッグオプション/スタックチェック/トレースを有効にします
  -debug                 すべての実行時エラーチェックを有功にします
  -o <ファイル>          出力先を <ファイル> にします
  -b                     すべての入力ファイルをひとつに結合します
                         動的ロード可能モジュール
  -E                     前処理のみ; コンパイルやリンクを行いません
  -C                     トランスレートのみ; COBOL から C へ変換します
  -S                     コンパイルのみ; アセンブリファイルを出力します
  -c                     コンパイルとアセンブルを行い、リンクを行いません
  -t <ファイル>          プログラムリストを <ファイル> に生成します
  -I <ディレクトリ>      COPY/INCLUDEの探索パスに <ディレクトリ> を加えます
  -L <ディレクトリ>      ライブラリの探索パスに <ディレクトリ> を加えます
```

```
      -l <lib>                ライブラリ <lib> をリンクします
      -B <options>            Cコンパイルフェーズに <options> を追加します
      -Q <options>            Cリンクフェーズに <options> を追加します
      -D <define>             Cコンパイラに <define> を渡します
      -conf=<ファイル>        ユーザ定義の方言設定 - -std=を参照してください
      --list-reserved         予約語の一覧を表示します
      --list-intrinsics       組み込み関数の一覧を表示します
      --list-mnemonics        作成者語の一覧を表示します
      -save-temps(=<dir>)     中間生成ファイルを保存します（デフォルトはカレントディレクトリ）
      -MT <target>            依存関係リストで使用される対象ファイルを指定します
      -MF <ファイル>          依存関係リストを <ファイル> に生成します
      -ext <extension>        既定のファイル拡張子を追加します
      -assign_external        すべてのASSIGN句に省略値EXTERNALが指定されたとみなします
      -reference_check        実行時の参照チェックを有効にします
      -constant(=<name"value">) $IF 文で評価する定数名 <name> に 値 <value> を設定します

      -W                      すべての警告を有功にする
      -Wall                   以下を除くすべての警告を有功にする
      -Wobsolete              廃要素が使われていれば警告する
      -Warchaic               古い仕様が使われていれば警告する
      -Wredefinition          データ項目の再定義を警告する
      -Wconstant              不適切な定数を警告する
      -Wparentheses           OR と AND が括弧なしで並んでいれば警告する
      -Wstrict-typing         タイプの不適合を厳密に警告する
      -Wimplicit-define       データ項目の再定義を警告する
      -Wcall-params           CALLのパラメタに指定された01レベルおよび77レベル以外の項目を警告する
                                (-Wall指定時は適用されません)
      -Wcolumn-overflow       72 桁を越えるテキストを警告する(-Wall指定時は適用されません)
      -Wterminator            終止符(END-XXX)がなければ警告する(-Wall指定時は適用されません)
      -Wtruncate              項目の切り詰めの可能性を警告する(-Wall指定時は適用されません)
      -Wlinkage               使われない連絡節項目を警告する(-Wall指定時は適用されません)
      -Wunreachable           実行されない文を警告する(-Wall指定時は適用されません)
      -Wcompat                コンパイラ実装間で非互換を発生しやすい記述を警告する(-Wall指定時は適用されません)

      -ftrace                 トレースコードの生成(実行された節/段落の追跡)
      -ftraceall              トレースコードの生成(実行された節/段落/文の追跡)
      -fsyntax-only           文法チェックのみ。何も出力しない
      -fdebugging-line        デバッグ行(標識領域に'D')を有効にする
      -fsource-location       ソース行情報の生成(-debugか-gで有効)
      -fimplicit-init         Cobolラインタイム初期化の自動実行
      -fsign-ascii            ASCII符号で数字を表示(ASCII機のデフォルト)
      -fsign-ebcdic           EBCDIC符号で数字を表示(EBCDIC機のデフォルト)
      -fstack-check           PERFORM実行スタックのランタイムチェック(-debugまたは-gで有効)
      -ffold-copy-lower       COPYブック名の小文字化(デフォルトは変換なし)
      -ffold-copy-upper       COPYブック名の大文字化(デフォルトは変換なし)
      -fnotrunc               2進項目のPICTURE句に合わせた切り詰めを行わない
      -ffunctions-all         組み込み関数使用時のFUNCTIONキーワードの省略を許す
      -fmfcomment             第1カラムの'*'と'/'をコメント行標識と解釈する(固定形式のみ)
      -fnull-param            CALL文のパラメタにNULL終端ポインタを追加して受け渡す
```

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。

8.1.3. 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。

8.1.4. 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. 「xxxxxxxx.cbl」または「xxxxxxxx.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「xxxxxxxx.so」、Windowsシステムでは「xxxxxxxx.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「xxxxxxxx.so」(UNIX)または「xxxxxxxx.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'xxxxxxxx'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、8.2.2で説明しているように、cobcrunコマンドを使う必要がある。

8.1.5. 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。

8.1.6. 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。

8.1.6.1. 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。

8.1.6.2. 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。

8.1.6.3. 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8-1-CまたはopensourceCOBOLのデータ型の一致

COBOLのUSAGE句 (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
BIARY-CHAR BINARY-CHAR UNSIGNED	1バイト	0 ~ 255	unsigned char
BINARY-CHAR SIGNED	1バイト	-128 ~ +127	signed char
BINARY-SHORT BINARY-SHORT UNSIGNED	2バイト	0 ~ 65535	unsigned unsigned int

			unsigned short unsigned short int
BINARY-SHORT SIGNED	2バイト	-32768 ~ +32767	int short short int signed int signed short signed short int
BINARY-LONG BINARY-LONG UNSIGNED	4バイト	0 ~ 4294967295	unsigned long unsigned long int
BINARY-LONG SIGNED	4バイト	-2147483648 ~ +2147483647	long long int signed long signed long int
BINARY-C-LONG SIGNED	4バイトまたは8バイト	-2147483648 ~ +2147483647または-9223372036854775808 ~ +9223372036854775807	long(USAGE BINARY-C-LONGの表5-10を参照)
BINARY-DOUBLE BINARY-DOUBLE UNSIGNED	8バイト	0 ~ 18446744073709551615	unsigned long long unsigned long long int
BINARY-DOUBLE SIGNED	8バイト	-9223372036854775808 ~ +9223372036854775807	long long int signed long long int
COMPUTATIONAL-1	4バイト	-3.4×10^{38} ~ $+3.4\times10^{38}$ (小数点以下6桁の精度)	float
COMPUTATIONAL-2	8バイト	-1.7×10^{308} ~ $+1.7\times10^{308}$ (小数点以下15桁の精度)	double
N/A(opensource COBOLに相当するものなし)	12バイト	-1.19×10^4932 ~ +1.19×10^4932 (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)。

8.1.6.4. 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8-2-opensource COBOLのC呼び出し

(maincob.cbl)
opensource COBOL メインプログラム

```
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.
```

(subc.c)
呼び出される C サブプログラム

```
#include <stdio.h>

int subc(char *arg1,
         char *arg2,
         unsigned long *arg3) {
 char nu1[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;
}
```

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。 `30`

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」データ項目にあるという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。

8.1.6.5. CメインプログラムのopensourceCOBOL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8-3-Cのopensource COBOL呼び出し

(mainc.c)
Cメインプログラム

```
#include <libcob.h>
#include <stdio.h>

int main (int argc, char **argv) {
 int returnCode;
 char arg1[7] = "Arg1";
 char arg2[7] = "Arg2";
 unsigned long arg3 = 123456789;
 printf("Starting mainc...\n");
 cob_init (argc, argv);
/* cob_init(0,NULL) if cmdline args not
going
to COBOL */
 returnCode = subcob(arg1,arg2,&arg3);
 printf("Back\n");
 printf("Arg1=%s\n",arg1);
 printf("Arg2=%s\n",arg2);
 printf("Arg3=%d\n",arg3);
 printf("Returned value=%d\n",returnCode);
 return returnCode;
}
```

(subcob.cbl)
呼び出される opensource COBOL サブプログラム

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. subcob.
 DATA DIVISION.
 LINKAGE SECTION.
 01 Arg1             PIC X(7).
 01 Arg2             PIC X(7).
 01 Arg3             USAGE BINARY-LONG.
 PROCEDURE DIVISION USING
     BY VALUE     Arg1,
     BY REFERENCE Arg2,
     BY REFERENCE Arg3.
 000-Main.
     DISPLAY 'Starting cobsub.cbl'.
     DISPLAY 'Arg1=' Arg1.
     DISPLAY 'Arg2=' Arg2.
     DISPLAY 'Arg3=' Arg3.
     MOVE 'X' TO Arg1 (1:1).
     MOVE 'X' TO Arg2 (1:1).
     MOVE 987654321 TO Arg3.
     MOVE 2 TO RETURN-CODE.
     GOBACK.
```

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 `31` またはBY VALUE句 `32` のいずれかで渡すことができる。

`30` 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。

`31` 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。

`32` 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。

8.1.7. 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使用できる様々な環境変数を示している。

表8-4-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。 **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[33]	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「 **-Iprefix/include** 」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、8.1.9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、8.1.8で説明する。
COB_IO_ASSUME_REWRITE	この環境変数に「Y」を設定することで、I-Oオプションでファイルを開いた時のWRITEをREWRITEに読み替えられるようにする。
COB_LDADD	プログラムとリンクする必要のある標準ライブラリが見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のある標準ライブラリが見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「 **-Lprefix/lib-lcob** 」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。
COB_NIBBLE_C_UNSIGNED	この環境変数に「Y」を設定することで、字類検査においてPIC 9項目の値に符号ニブル「C」を許容する。
COB_VERBOSE	この環境変数に「Y」を設定することで、SORT実行時に出力するメッセージを冗長化することが可能になる。
COBCPY	この環境変数は、コンパイラがCOPYモジュールを見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、8.1.8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。
OC_EXTEND_CREATES	この環境変数に「yes」を設定することで、EXTENDオプションでファイルを開く時に自動でファイルが生成される。

OC_IO_CREATES	この環境変数に「yes」を設定することで、I-Oオプションでファイルを開く時に自動でファイルが生成される。
OC_USERFH	この環境変数にCOBOLプログラム名を指定することで、COBOLのファイル処理をユーザ定義のプログラムで実行できるようになる。OPEN, CLOSE, DELETE, READ, REWRITE, START, WRITE, COMMIT, ROLLBACK, UNLOCKの処理がサポートされている。
TMPDIR TMP (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)。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。別の一時フォ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。

---

33 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。

8.1.8. 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ると終了する。

- 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(8.1.2を参照)で指定されたフォルダ。
- COBCPY環境変数(8.1.7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。 34 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(8.1.7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
- 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。

---

34 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文字(:)を使用する。

8.1.9. 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 - 2     1
#                   3 - 4    3 - 4     2
#                   5 - 6    5 - 7     3
#                   7 - 9    8 - 9     4
#                  10 - 11  10 - 12    5
#                  12 - 14  13 - 14    6
#                  15 - 16  15 - 16    7
#                  17 - 18  17 - 18    8
binary-size: 1-2-4-8

# 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
```

8.2. opensource COBOLプログラムの実行

8.2.1. 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
```

8.2.2. 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(8.1.4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。

8.2.3. 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

8.2.4. 重要な環境変数

次の表は、opensource COBOLプログラムの実行で使用できる様々な環境変数を示している。

表8-5-実行時環境変数

環境変数	使い方
COB_DATE	システム日付に任意の日付を「yyyy/mm/dd」の形式で設定する。
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、すぐに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[35]。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[36]。
PATH	opensource COBOLの「bin」ディレクトリはPATHで定義する必要がある。

TMPDIR TMP TEMP (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。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[37]。

35 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。

36 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。

37 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。

8.3. 組み込みサブルーチン

8.3.1. 「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
- 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。

8.3.1.1. CALL “C$CALLEDBY” USING *program-name GIVING status*

このルーチンは、実行中のCOBOLプログラムを呼出したプログラム名を返す。呼出しプログラムが存在しないか未知の場合には、空白を戻す。

*program-name*には呼出しプログラム名か、呼出しプログラムが存在しないか未知の場合には空白を含む。呼出されたプログラムがオブジェクトライブラリにあると、プログラムはPROGRAM-IDを戻す。オブジェクトライブラリにもないと、ディスク名が戻される。

statusは次のいずれかの値を受け取る。

値	説明
1	ルーチンは他のCOBOLプログラムによって呼出された。

0	ルーチンは主プログラムである。呼出しプログラムは存在しない。
-1	呼出しプログラムは未知である。ルーチンはCOBOLプログラムから呼出されたのではない。

8.3.1.2. 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。

8.3.1.15章—**CBL_CHANGE_DIR**を参照

8.3.1.3. 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。

8.3.1.18章—**CBL_COPY_FILE**を参照

8.3.1.4. 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。

8.3.1.22章—**CBL_DELETE_FILE**を参照

8.3.1.5. 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 `38` 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 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。

8.3.1.16章—**CBL_CHECK_FILE_EXIST**を参照

8.3.1.6. 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

8.3.1.7. CALL “C$LIST-DIRECTORY” USING *item-1, item-2, item-3*

このルーチンは、選択されたディレクトリの内容をリストする。各オペレーティングシステムには、このタスクを果たす独特の方法がある。C$LIST-DIRECTORYは、すべてのオペレーティングシステムのために機能する一つの方法を提供する。

与えられたディレクトリにあるファイルの名前を取得することを可能にする。3つの明白な操作によってこれを成し遂げる。最初の操作は指定されたディレクトリを開き、そして、ファイルのリストを作成する。第2の操作で1つずつリストにあるファイル名を返し、第3の操作でディレクトリを閉じ、ルーチンによって使われた全てのメモリを解放する。

item-1の値	説明
1の時	指定されたディレクトリを開く。item-2にはDIRECTORY、item-3にはPATTERNを設定する。
2の時	開かれたディレクトリからファイル名を読み取る。item-2にはMYDIR、item-3にはFILENAMEを設定する。
3の時	他の操作によって使用された資源を解放する。メモリ漏洩を回避するために、呼ばれなければならない。item-2にはLISTDIR-NEXT操作で設定するデータ項目と同じものを設定する。

```
01 PATTERN      PIC X(5) VALUE "*".
01 DIRECTORY    PIC X(256) VALUE
"./list".
01 FILENAME     PIC X(30).
01 MYDIR        PIC 9(8) COMP-5.
PROCEDURE      DIVISION.
    CALL "C$LIST-DIRECTORY" USING 1,
                                  DIRECTORY,
                                  PATTERN
    END-CALL.
    MOVE RETURN-CODE TO MYDIR.
    CALL "C$LIST-DIRECTORY" USING 2,
                                  MYDIR,
                                  FILENAME
    END-CALL.
    PERFORM WITH TEST AFTER UNTIL FILENAME = SPACES
      DISPLAY FUNCTION TRIM(FILENAME)
      CALL "C$LIST-DIRECTORY" USING 2,
                                    MYDIR,
                                    FILENAME
      END-CALL
    END-PERFORM.
    CALL "C$LIST-DIRECTORY" USING 3, MYDIR
    END-CALL.
```

8.3.1.8. 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。

8.3.1.19章—**CBL_CREATE_DIR**を参照

8.3.1.9. 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。

6.1.7章—**NUMBER-OF-CALL-PARAMETERS**を参照

8.3.1.10. 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。

8.3.1.11. 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。

8.3.1.33章—**CBL_OC_NANOSLEEP**を参照

8.3.1.12. 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。

8.3.1.38章—**CBL_TOLOWER**を参照

8.3.1.13. 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。

8.3.1.39章—**CBL_TOUPPER**を参照

8.3.1.14. 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**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。

下の真理値表は「AND」プロセスを示している。

引数1ビット	引数2ビット	新しい引数2ビット
0	0	0
0	1	0
1	0	0
1	1	1

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。

8.3.1.15. 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。

8.3.1.2章—**C$CHDIR**を参照

8.3.1.16. 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 `39` 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。

8.3.1.5章—**C$FILEINFO**を参照

8.3.1.17. 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。

8.3.1.18. 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。

8.3.1.3章—**C$COPY**を参照

8.3.1.19. 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。

8.3.1.8章—**C$MAKEDIR**を参照

8.3.1.20. 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。`40`

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。

8.3.1.34章—**CBL_OPEN_FILE**を参照

8.3.1.21. 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。

8.3.1.22. 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。

8.3.1.4章—**C$DELETE**を参照

8.3.1.23. 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
            '*** ' Formatted-Date ' ' Formatted-Time ' ***'
        END-DISPLAY
        GOBACK
        .
```

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'
```

8.3.1.24. 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
          '*** ' Formatted-Date ' ' Formatted-Time ' ***'
      END-DISPLAY
      GOBACK
      .
```

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'
```

8.3.1.25. 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**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**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。

下の真理値表は「EQ」プロセスを示している。

引数1ビット	引数2ビット	新しい引数2ビット
0	0	1
0	1	0
1	0	0
1	1	1

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。

8.3.1.26. 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。

8.3.1.27. 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0length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。

8.3.1.28. 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**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。

下の真理値表は「IMP」プロセスを示している。

引数1ビット	引数2ビット	新しい引数2ビット
0	0	1
0	1	1
1	0	0
1	1	1

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。

8.3.1.29. 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。

下の真理値表は「NIMP」プロセスを示している。

引数1ビット	引数2ビット	新しい引数2ビット
0	0	0
0	1	0
1	0	1
1	1	0

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。

8.3.1.30. 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。

下の真理値表は「NOR」プロセスを示している。

引数1ビット	引数2ビット	新しい引数2ビット
0	0	1
0	1	0
1	0	0
1	1	0

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。

8.3.1.31. 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**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**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。

下の真理値表は「NOT」プロセスを示している。

古い引数2ビット	新しい引数2ビット
0	1
1	0

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。

8.3.1.32. CALL “CBL_OC_KEISEN” USING *item-1*

CBL_OC_KEISENは、画面に縦・横の罫線を表示することができる。

item-1として次の集団項目を定義する。

```
    01 KEISEN.
      02 KEI-CMD   PIC 9(1) COMP-X.
      02 KEI-LINE  PIC 9(2) COMP-X.
      02 KEI-COL   PIC 9(2) COMP-X.
      02 KEI-LNG1  PIC 9(2) COMP-X.
      02 KEI-LNG2  PIC 9(2) COMP-X.
      02 KEI-COLOR PIC 9(2) COMP-X.
      02 KEI-PRN   PIC 9(2) COMP-X.
```

各項目の意味を以下に示す。

項目	意味
KEI-CMD	0-初期設定(画面消去) 1-アンダーライン(下) 2-オーバーライン(上)

	3-バーティカルライン(左) 4-バーティカルライン(右) 5-ボックス 6-バーティカル(左)とアンダーライン(下) 9-終了処理
KEI-LINE	開始ライン(1～24)
KEI-COL	開始カラム(1～80)
KEI-LNG1	線長 横線(1～80) KEI-CMD：1、2、5 縦線(1～24) KEI-CMD：3、4
KEI-LNG2	線長 縦線(1～24) KEI-CMD：5
KEI-COLOR	線の色 0-黒 1-青 2-緑 3-青緑 4-赤 5-深紅 6-茶 7-白 モノクロ端末では、白に設定される。
KEI-PTN	線種 1-実線 2-破線 3-点線 4-一点鎖線 5-二点鎖線

8.3.1.33. 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。

8.3.1.11章—**C$SLEEP**を参照

8.3.1.34. 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使用される。

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。

8.3.1.20章—**CBL_CREATE_FILE**を参照

8.3.1.35. 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。

下の真理値表は「OR」プロセスを示している。

引数1ビット	引数2ビット	新しい引数2ビット
0	0	0
0	1	1
1	0	1
1	1	1

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。

8.3.1.36. 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。`41` 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。

8.3.1.37. 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。

8.3.1.38. 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。

8.3.1.12章—**C$TOLOWER**を参照

8.3.1.39. 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。

8.3.1.13章—**C$TOUPPER**を参照

8.3.1.40. 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。

8.3.1.41. 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。

下の真理値表は「XOR」プロセスを示している。

引数1ビット	引数2ビット	新しい引数2ビット
0	0	0
0	1	1
1	0	1
1	1	0

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。

8.3.1.42. 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。

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。

---

38 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。

39 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。

40 **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。

41 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。

9. サンプルプログラム

9.1. 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.
```

9.2. 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、ユーティリティサブルーチンである。

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBDUMP.
*****************************************************************
** This is an OpenCOBOL 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.c                              **
**                                                             **
** 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 OpenCOBOL 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
.
```

クレジット

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

【マークダウン化】
東京システムハウス株式会社 馮婉怡、横川桃子、横山颯斗

【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/