J. A. C

COMMODORE SUPERPET COMPUTERS

Waterloo microCOBOL©

commodore COMPUTER

Waterloo microCOBOL

Tutorial

and

Reference Manual

P.H.Dirksen

J.W.Welch

Disclaimer

Preface

Waterloo microCOBOL is intended to be a substantial implementation of the standard COBOL language. The language supported is suitable for both teaching purposes and for programming many business problems.

It is intended to make available a number of different microCOBOL processors. At the time of writing, interpreters are available for the Commodore SuperPET and for the IBM VM/CMS operating system. Interpreters are being tested for the IBM Personal Computer and for DEC VAX VMS systems. As well, compilers are being developed for the systems mentioned.

This manual is presented in two parts. The first part is a collection of annotated examples intended to introduce the reader to many of the features of microCOBOL. In this way, a novice is presented with a staged introduction to the language. An experienced programmer could use the examples to compare microCOBOL to other COBOL implementations or to other languages.

The second part is a comprehensive language reference manual for Waterloo microCOBOL. Essentially, the language supported includes level one of the **NUCLEUS, SEQUENTIAL I-O, RELATIVE I-O** and **TABLE HANDLING** modules described in COBOL Standards (ANSI X3.23-1974 or ISO 1989-1978). Parts of level two in these modules have also been implemented, including full support for the **PERFORM, STRING** and **UNSTRING** verbs. A few items have been omitted from level one:

(1) The **I-O-CONTROL** paragraph in the **ENVIRONMENT DIVISION** is not supported.

(2) The **DELETE** statement in **RELATIVE I-O** is not supported.

(3) Paragraph and section names must contain at least one alphabetic character.

(4) Continuation of a line is not supported. Syntactic units, such as data names or literals cannot be split across lines.

No support is provided for tape hardware.

P. H. Dirksen
J. W. Welch

April 1982

Acknowledgement

The design and implementation of the Waterloo microCOBOL processors is based upon ideas evolved over the past decade in a number of organizations. All members of the Computer Systems Group (University of Waterloo), the Waterloo Foundation for the Advancement of Computing, and Waterloo Computing Systems, Ltd. have made a substantial contribution to its development. The actual design and programming of the system directly involved the following people: James Bruyn, Keith Campbell, Martin Leistner, Lyle Resnick, Liz Ruest, Jack Schueler, David Till and Jim Welch. Sharon Haydamak was responsible for the production of this manuscript.

Table of Contents

Waterloo microCOBOL

Tutorial Examples

Much of the material in the tutorial portion of this text is the result of experience accumulated over many years of presenting courses on the subject of file processing at University of Waterloo. In particular, the authors wish to acknowledge the work done in the following text.

An Introduction to COBOL with WATBOL,
A Structured Programming Approach
D.D.Cowan, P.H.Dirksen, and J.W.Graham,
WATFAC Publications,
Box 803,
Waterloo, Ontario, Canada.

Waterloo Computing Systems Newsletter

The software described in this manual was implemented by Waterloo Computing Systems Limited. From time-to-time enhancements to this system or completely new systems will become available.

A newsletter is published periodically to inform users of recent developments in Waterloo software. This publication is the most direct means of communicating up-to-date information to the various users. Details regarding subscriptions to this newsletter may be obtained by writing:

Waterloo Computing Systems Newsletter
Box 943,
Waterloo, Ontario, Canada
N2J 4C3

Chapter 1

Tutorial Examples

1.1 Introduction

The following tutorial is a sequence of examples meant to introduce the reader to the "flavour" of Waterloo microCOBOL. They do not present a complete or rigorous treatment of any topic, as this detailed information is available in the reference manual in the latter part of this document. This tutorial could be useful in the following situations:

(1) Someone already familiar with COBOL can determine some of the major differences between Waterloo microCOBOL and the dialect already known.

(2) Teachers may find the examples useful as a progressive introduction of the material to their students.

(3) People who already know some other language can get an appreciation for Waterloo microCOBOL before reading the reference manual.

(4) Complete novices could run the various programs, and possibly learn some of the material by exploring the various language features in conjunction with the reference material.

In order that the examples be fully appreciated, it is important that they be entered into the computer and executed.

1.2 Introductory Examples.

1.2.1 A Minimum Program.

Every COBOL program requires a certain number of basic statements. These are presented in this example.

```
identification division.
program-id. EXAMPLE-1.
environment division.
configuration section.
source-computer. CBM-SuperPET.
object-computer. CBM-SuperPET.
data division.
procedure division.
    stop run.
```

Notes

(1) COBOL is a programming language which was first developed in the early 1960's to be used to solve business data processing problems. COBOL in fact stands for **CO**mmon **B**usiness **O**riented **L**anguage.

(2) In order to run any COBOL program a number of statements are required. These statements must be present for the program to work properly.

(3) Example 1 contains these basic necessary statements. All future examples will also contain these statements with some possible slight modifications and additions.

Hint The reader should enter these lines exactly as they appear. This set of statements can then be saved in a file. Thus they need not be entered for each program but instead can be retrieved using the **get** command. In future examples, this file will be referred to as "texta". The following notes should be read before one enters the above lines.

(4) COBOL statements are usually entered beginning in either column 2 or column 6.

(5) The first seven statements in this example are entered in column 2 while the last statement is entered in column 6.

(6) Column 2 is called *margin* A and to column 6 is called *margin* B.

(7) Each line ends with a period.

(8) Users who have used COBOL before will notice that the programs are entered in lower case letters. Waterloo microCOBOL permits the use of both upper and lower case letters; the rules when upper and lower case letters are both used are described in the reference manual.

(9) If the reader wishes to know more about these statements, he should refer to the reference manual portion of this text. As more examples are presented, these statements will be described in more detail. However, the following notes about these statements are appropriate.

Notes

(1) A COBOL program consists of four divisions. These are the identification division, the environment division, the data division and the procedure division. Note that each of these appears once in this example.

(2) The use of 'EXAMPLE-1' as a **program-id** is for documentation purposes only.

(3) Users will recognize 'CBM-SuperPET' as the name of a computer. If programs are run on a different system, it is not necessary to change this name as it is only used for documentation purposes.

1.2.2 Display a name.

One of the first things we want to do in a program is to display information. This program demonstrates one simple way to accomplish this.

```
*
* Display a Name on the Terminal.
*
 identification division.
 program-id. EXAMPLE-2.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.
 procedure division.
      display 'James'.
      stop run.
```

Sample Program Execution

```
run
Execution begins...
James
...Execution ends.
```

Notes

(1) In this example and all future examples the output produced by the program is displayed following the program. A line appearing in *italics* represents a line entered by the user; non-italicized lines have been displayed by the COBOL processor. When a program is *run*, at least two lines are displayed, namely that the program has started and that the program has stopped. This is indicated by the lines

```
Execution begins...
```

and

```
...Execution ends.
```

(2) The line containing the name

```
James
```

is displayed between the above two lines. It is the output produced by the program. Any output will always appear between these two lines.

(3) Three lines have been inserted at the beginning of the program. These lines, containing an * (asterisk) in column 1, are called *comments*. Comments have no effect on the program; they are used for documentation purposes only. Comment lines may be entered anywhere in the program.

(4) The line

```
display 'James'.
```

has been inserted in the procedure division portion of the program. When the program is run, this causes the line containing the name James to be displayed.

(5) The characters to be displayed are enclosed by quotation marks.

(6) The procedure division now contains two statements which are more commonly referred to as *sentences*. Each sentence begins with a *verb* which indicates the desired action to be performed. Each sentence ends with a period.

(7) The **display** verb causes the string of characters to be displayed on the screen.

(8) The **stop** verb indicates that no more actions are required.

(9) The **program-id** line has been used to give a different name to this program. It is used for documentation purposes only.

(10) It is important to note that every program processed goes through two distinct phases, one following the other in time. First the program is read by the system to determine certain types of errors, in particular syntax or grammar errors are detected. Then the system actually begins executing the statements.

(11) This program could be easily entered by using the following steps:

i) Use the **get** command to load a copy of the previous program.

ii) Add the three **comment** lines at the top of the program.

iii) Add the **display** statement.

Use of this technique results in less time to enter the program and also reduces "entry" errors. As we will see in future examples, it is often easier to modify an existing program than to enter completely a new program.

1.2.3 Accept Data from the Terminal.

Another common requirement is to input some information into a program. This program accepts some data and then displays it on the screen.

```
*
* Accept Data from the Terminal.
*
 identification division.
 program-id. EXAMPLE-3.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 procedure division.
      display 'Enter a name of 5 characters'.
      accept name.
      display name.
      stop run.
```

Sample Program Execution

```
run
Execution begins...
Enter a name of 5 characters
James
James
...Execution ends.
```

Notes

(1) When this program is run, a line is displayed asking that a name containing exactly 5 characters be entered. The next line, which appears in italics, is the response entered by the user, namely the name James. The following line is the output produced by the program. In this and all future examples, all lines entered by the user are displayed in italics.

(2) The data read from the terminal is placed in **working-storage section** of the **data division.** Working storage can be thought of a large piece of paper within the computer. Information is "written" or placed into working storage.

(3) For this particular example, working storage consists of an area large enough to contain a 5-character string which will be called "name". It is defined as follows:

```
working-storage section.
01  name picture xxxxx.
```

The 01 is a *level number* and is entered in margin A. "Name" is the name of the area and is entered in margin B. The **picture** clause defines the characteristics of the area. In this case, an area capable of holding 5 consecutive characters is defined by using five x's.

(4) Level numbers will be explained more fully in future examples.

(5) "Name" is referred to as a *data-name* and is chosen by the programmer. The rules for choosing such names are described in a later note.

(6) The **accept** verb is used to input the desired string of characters, specifying it should be placed in working-storage in the area called "name".

(7) The **accept** verb takes the characters that are entered and places them in the area called "name". If more than 5 characters are entered the left-most 5 characters are placed in "name". If less than 5 characters are entered, the characters are placed left-justified in "name". The remaining characters are left unchanged. Thus the user should enter blanks or spaces if the name contains less than 5 characters. The next example shows another way of accepting variable length names.

(8) Blank lines are inserted before and after the working-storage section to make the program easier to read.

(9) A data-name consists of not more than thirty characters chosen from the letters, the digits, and the hyphen; it must contain at least one letter. The hyphen may be used anywhere except at the beginning or end.

(10) COBOL reserves a number of words for its own use. These are called *reserved words*. For example, all COBOL verbs are reserved words as are most of the words in "texta". COBOL's reserved words are listed in the reference section of this text (see RESERVED WORDS).

(11) A data-name *cannot* be a COBOL reserved word. For example, it is not possible to use the data-name "input" instead of "name" since "input" is a reserved word. However, it is possible to use "input-place".

1.2.4 The Perform Verb.

Programs can be written in such a way that they are easier to read and understand. This clarity is achieved by organizing the program into modules or parts.

```
*
* Introduce Perform Verb.
*
 identification division.
 program-id. EXAMPLE-4.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 procedure division.
     perform get-name.
     perform display-name.
     stop run.

 get-name.
     display 'Enter a name up to 5 characters'.
     move spaces to name.
     accept name.

 display-name.
     display name.
```

Sample Program Execution

```
run
Execution begins...
Enter a name up to 5 characters
Jim
Jim
...Execution ends.
```

Notes

(1) This program is a slight modification of the previous example. The procedure division is organized differently.

(2) The two actions of accepting the name to be read and displaying the name have been separated into two distinct parts or modules.

(3) In COBOL, these parts are called *paragraphs;* each paragraph is identified by a *paragraph-name*. The paragraph-name is entered in margin A and the paragraphs are placed following the **stop run.**

(4) Paragraph-names are formed in the same way as data-names. When a paragraph-name is used to signify the beginning of a paragraph, it must be followed by a period.

(5) Blank lines have been inserted to make it easier to identify the paragraphs.

(6) The **perform** verb in the sentence

```
perform get-name
```

acts exactly as one would expect, namely it causes the sentences in the paragraph named "get-name" to be executed.

(7) When the "get-name" paragraph is completed, control passes to the sentence following the

```
perform get-name
```

namely the

```
perform display-name.
```

(8) The two **performs** cause the two paragraphs to be executed in the appropriate sequence.

(9) The two paragraphs can be placed in any order following the **stop run.** Of course, the two **performs** must be placed in the correct sequence in order for the program to function properly.

(10) A new sentence

```
move spaces to name
```

has been inserted before the **accept** sentence. This will cause five spaces or blank characters to be placed in "name". We can now enter a name of any length up to 5 characters. The **move** verb will be described in more detail in a future example.

1.2.5 The Until Clause.

One of the most important features of computers is to perform certain tasks a number of times. This program introduces one method of doing such tasks repetitively until a signal is encountered to stop the process.

```
*
* Perform Verb with Until Clause.
*
 identification division.
 program-id. EXAMPLE-5.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 procedure division.
     perform get-name.
     perform process-name
         until name = 'stop '.
     stop run.

 get-name.
     display 'Enter a name up to 5 characters'.
     move spaces to name.
     accept name.

 process-name.
     display name.
     perform get-name.
```

Sample Program Execution

```
run
Execution begins...
Enter a name up to 5 characters
James
James
Enter a name up to 5 characters
Jim
Jim
Enter a name up to 5 characters
Mary
Mary
Enter a name up to 5 characters
stop
...Execution ends.
```

(1) This program asks the user to enter a name. After the name is displayed, the program asks that another name be entered. This process continues until the characters 'stop ' are entered at which time the program terminates.

(2) The program is written to accept and then display an unknown number of names. The two actions of displaying and reading the name, are placed in a paragraph called "process-name".

(3) The "process-name" paragraph makes use of the previously written paragraph "get-name", which displays the prompt message and then reads a name. Paragraphs can contain **performs** of other paragraphs.

(4) The program now works as follows:

- i) An initial name is read using the "get-name" paragraph.
- ii) The "process-name" paragraph is then performed. This causes the name to be displayed and another name to be read.
- iii) Control returns to the **perform-until** sentence which determines if the newly entered string is 'stop '. If not, the "process-name" paragraph is executed again. If the name is 'stop ', control passes to the sentence following the **perform-until.**

(5) We refer to

```
name = 'stop '
```

as a *condition*, in this case the *equals* condition. The condition compares the value of "name" with the characters 'stop '. If they are equal the value of the condition is true; otherwise it is false. A more general form of condition is discussed in a later section.

(6) While the clause

```
until name = 'stop '
```

could have been entered on the same line as the **perform,** it has been entered as a separate line and indented to make the program more readable.

(7) A COBOL sentence can be written on more than one line. Sometimes this occurs because a line is too long but more often it is done to improve readability. The continued line is usually indented in order that the continuation can be clearly seen.

(8) The condition could have been written as

```
name = 'stop'
```

In this case, the blank character has been omitted from the end of the string. Before comparing two fields COBOL checks that the two strings have the same length. If one string is shorter, it is padded on the right with blank characters to make it the same length as the longer string. Thus 'stop' would be set to 'stop ' before the comparison is done.

1.2.6 Read a Number of Fields from the Terminal.

On many occasions we wish to enter a number of items of information about a particular person or thing. For example, we might wish to also enter such items as sex, age, etc.

```
*
* Read a Number of Fields from the Terminal.
*
 identification division.
 program-id. EXAMPLE-6.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 data division.

 working-storage section.

 01   student-no             pic xxxx.
 01   name                   pic xxxxx.
 01   age                    pic xx.
 01   sex                    pic x.

 procedure division.
      perform get-id.
      perform process-student-data
           until student-no = '9999'.
      stop run.

 process-student-data.
      perform get-name.
      perform get-age.
      perform get-sex.
      display student-no name age sex.
      perform get-id.

 get-id.
      display 'Enter student number (9999 to stop)'.
      move spaces to student-no.
      accept student-no.
```

```
get-name.
    display 'Enter name'.
    move spaces to name.
    accept name.

get-age.
    display 'Enter age'.
    move spaces to age.
    accept age.

get-sex.
    display 'Enter sex (M or F)'.
    accept sex.
```

Sample Program Execution

```
run
Execution begins...
Enter student number (9999 to stop)
1234
Enter name
James
Enter age
15
Enter sex (M or F)
M
1234James15M
Enter student number (9999 to stop)
2345
Enter name
Marie
Enter age
15
Enter sex (M or F)
F
2345Marie15F
Enter student number (9999 to stop)
9999
...Execution ends.
```

Notes

(1) This program prompts the user to enter 4 quantities, namely a student number, name, age, and sex and then displays this data on the terminal. This sequence is repeated until the student number '9999' is entered.

(2) Three lines have been added to the working-storage section to define the areas for the three new items of data. The new data-names are "student-number", "age" and "sex" and they have a size of 4, 2 and 1 characters respectively.

(3) The reserved word **picture** is used quite frequently in COBOL programs. To save time and space, a short form, **pic** can be used.

(4) This program uses the student number '9999' to terminate processing instead of the name 'stop' as in the previous example. Thus

```
perform get-id
```

is used at the start of the procedure division instead of

```
perform get-name.
```

(5) The "process-name" paragraph has been replaced by the paragraph called "process-student-data". It causes the number, age, and sex to be read, displays the appropriate line and then reads the next student number. The sentence

```
perform process-student-data
     until student-number = '9999'
```

controls the reading and displaying of the student data.

(6) The sentence

```
display student-no name age sex.
```

displays the line on the terminal. The data-names in this sentence are separated by a blank.

(7) For each record read, a line is displayed. It is somewhat disturbing that there are no spaces between the four items of output. This can be remedied by using

```
display student-no ' ' name ' ' age ' ' sex.
```

which inserts the blank character between each item.

(8) The sentence which displays the student data could be replaced by the two sentences

```
display student-no.
display name ' ' age ' ' sex.
```

which would cause two lines to be displayed for each student.

(9) More blanks could be placed between any of the items on the output line by increasing the number of spaces between the quotes.

```
display student-no '    ' name ...
```

would place 4 spaces between the number and the name.

(10) The four data items are quite often referred to as *fields*.

1.2.7 Define a Simple Data Structure.

In many cases it is more convenient to represent and deal with a number of data items or fields as a single entity. This collection of information about a particular person or thing is called a *record*.

```
*
* Read a Number of Fields from the Terminal.
*
 identification division.
 program-id. EXAMPLE-7.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 data division.

 working-storage section.

 01   student-data.
      02  student-no            pic xxxx.
      02  name                  pic xxxxx.
      02  age                   pic xx.
      02  sex                   pic x.

 procedure division.
      perform get-id.
      perform process-student-data
           until student-no = '9999'.
      stop run.

 process-student-data.
      perform get-name.
      perform get-age.
      perform get-sex.
      display student-data.
      perform get-id.

 get-id.
      display 'Enter student number (9999 to stop)'.
      move spaces to student-no.
      accept student-no.
```

```
get-name.
      display 'Enter name'.
      move spaces to name.
      accept name.

get-age.
      display 'Enter age'.
      move spaces to age.
      accept age.

get-sex.
      display 'Enter sex (M or F)'.
      accept sex.
```

Sample Program Execution

```
run
Execution begins...
Enter student number (9999 to stop)
4321
Enter name
Fred
Enter age
16
Enter sex (M or F)
M
4321Fred 16M
Enter student number (9999 to stop)
6543
Enter name
Bev
Enter age
14
Enter sex (M or F)
F
6543Bev  14F
Enter student number (9999 to stop)
9999
...Execution ends.
```

Notes

(1) This program is another version of the previous example which creates a *data-structure* containing the four fields.

(2) The data structure is defined as follows:

```
01  student-data.
    02 student-no     pic xxxx.
    02 name           pic xxxxx.
    02 age            pic xx.
    02 sex            pic x.
```

A new 01-level data-name is introduced, namely "student-data". The four 01-level items from the previous example have been changed to 02-level items and have been entered in margin B.

(3) The data structure can be described as follows:

i) The four data-items can be considered as a collection of information about a particular student. This collection is called a *record* and the 01-level data-name permits the program to refer to the entire record.

ii) The original four data-items have the same data-names and the same sizes as before but they now have been defined with 02-level numbers. They can be used in the same way as they were used in previous examples.

(4) The 01-level item ends in a period.

(5) The 01-level item does not have a **picture** clause if it is subdivided into 02-level items.

(6) The 01-level item in this example is considered to have 12 characters i.e. the sum of the sizes of the fields at the 02-level.

(7) The user should again note that some or all of the output fields are not separated by blanks. A future example will remedy this situation.

1.2.8 Create an Output Data Structure.

Often when displaying a number of fields, we want to setup or format the line with appropriate spacing and then to refer to the line as an output record. An output data structure is defined and used in this example to allow more flexibility on output.

```
*
* Create an Output Data Structure.
*
 identification division.
 program-id. EXAMPLE-8.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 data division.

 working-storage section.

 01   student-data.
      02  student-no          pic xxxx.
      02  name                pic xxxxx.
      02  age                 pic xx.
      02  sex                 pic x.

 01   display-record.
      02  out-student-no      pic xxxx.
      02  filler              pic xxx value is spaces.
      02  out-name            pic xxxxx.
      02  filler              pic xxx value is spaces.
      02  out-age             pic xx.
      02  filler              pic xxx value is spaces.
      02  out-sex             pic x.

 procedure division.
      perform get-id.
      perform process-student-data
          until student-no = '9999'.
      stop run.
```

```
process-student-data.
    perform get-name.
    perform get-age.
    perform get-sex.
    perform edit-and-display-record.
    perform get-id.

get-id.
    display 'Enter student number (9999 to stop)'.
    move spaces to student-no.
    accept student-no.

get-name.
    display 'Enter name'.
    move spaces to name.
    accept name.

get-age.
    display 'Enter age'.
    move spaces to age.
    accept age.

get-sex.
    display 'Enter sex (M or F)'.
    accept sex.

edit-and-display-record.
    move student-no to out-student-no.
    move name to out-name.
    move age to out-age.
    move sex to out-sex.
    display display-record.
```

Sample Program Execution

```
run
Execution begins...
Enter student number (9999 to stop)
5555
Enter name
Eliza
Enter age
15
Enter sex (M or F)
F
5555   Eliza   15   F
Enter student number (9999 to stop)
1111
Enter name
Paul
Enter age
14
Enter sex (M or F)
M
1111   Paul    14   M
Enter student number (9999 to stop)
9999
...Execution ends.
```

Notes

(1) Each student record is read and then displayed with each field separated by at least three spaces.

(2) A new data structure called "display-record" is defined. It contains four fields with the newly defined data-names "out-student-no", "out-name", "out-age", and "out-sex". These will be used to contain the four fields for output.

(3) Each of these fields is separated by a field which is called **filler. Filler** is a COBOL reserved word and is used to "fill" or insert space in a record. The **picture** clause indicates how much space is to be inserted.

(4) The **value is** clause specifies the particular character or characters we wish to insert in the field. In this case, the COBOL reserved word **spaces** indicates that the field is to contain spaces or blanks.

(5) If the **value is** clause is omitted in the **filler,** the field is said to be *undefined*. If such a field were displayed, it would contain one or more question marks (??).

(6) The **move** verb is used to move an item from one place in working-storage to another. Thus

```
move student-no to out-student-no.
```

causes the 4-character number to be moved from "student-data" to "out-student-no" in "display-record".

(7) There is no difficulty with moving data from one field to another if the fields are the same size. However, if the *receiving* field is larger than the sending field, the data is inserted left-justified and an appropriate number of blanks are inserted on the right. The first two lines of the display record could be replaced by

```
02 out-student-no  pic xxxxxxx.
```

and the output would be the same since the four character name would be moved to the left-most four positions and blanks would be inserted in the remaining three positions.

(8) If the receiving field is smaller than the sending field, the data again is inserted left-justified. However, the 'extra' characters on the right are truncated.

1.2.9 Produce a Simple Report.

Definition of **picture** clauses can made simpler especially when 'long' fields are required.

```
*
* Read a Number of Fields from the Terminal
* and Print a Small Report.
*
 identification division.
 program-id. EXAMPLE-9.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 data division.

 working-storage section.

 01   student-data.
      02 student-no             pic xxxx.
      02 name                   pic xxxxx.
      02 age                    pic xx.
      02 sex                    pic x.

 01   heading-line.
      02 filler         pic x(12) value is 'Student Data'.

 01   first-line.
      02 filler                 pic x(8) value is 'number '.
      02 out-student-no         pic xxxx.

 01   second-line.
      02 filler                 pic x(5) value is 'name'.
      02 out-name               pic x(5).

 01   third-line.
      02 filler                 pic x(4) value is 'age '.
      02 out-age                pic xx.
      02 filler                 pic x(5) value is ' sex '.
      02 out-sex                pic x.
```

```
procedure division.
    perform get-id.
    perform process-student-data
        until student-no = '9999'.
    stop run.

process-student-data.
    perform get-name.
    perform get-age.
    perform get-sex.
    perform edit-and-display-record.
    perform get-id.

get-id.
    display 'Enter student number (9999 to stop)'.
    move spaces to student-no.
    accept student-no.

get-name.
    display 'Enter name'.
    move spaces to name.
    accept name.

get-age.
    display 'Enter age'.
    move spaces to age.
    accept age.

get-sex.
    display 'Enter sex (M or F)'.
    accept sex.

edit-and-display-record.
    move student-no to out-student-no.
    move name to out-name.
    move age to out-age.
    move sex to out-sex.
    display heading-line.
    display first-line.
    display second-line.
    display third-line.
```

Sample Program Execution

```
run
Execution begins...
Enter student number (9999 to stop)
9876
Enter name
Bob
Enter age
17
Enter sex (M or F)
M
Student Data
number  9876
name Bob
age 17 sex M
Enter student number (9999 to stop)
5786
Enter name
John
Enter age
16
Enter sex (M or F)
M
Student Data
number  5786
name John
age 16 sex M
Enter student number (9999 to stop)
9999
...Execution ends.
```

Notes

(1) The example contains most of the material that has been presented in the previous examples. The program prompts for a number, name, age, and sex and displays a small report for each student.

(2) A heading is placed at the beginning of each student report.

(3) The "heading-line" defines a field of 12 characters by using the clause

```
02 filler pic x(12) value is 'Student Data'.
```

which is equivalent to

```
02 filler pic xxxxxxxxxxxx value is 'Student Data'.
```

The former method of defining **pictures** is often more convenient than the latter.

(4) Both methods of defining **pictures** can be used in a particular data-structure definition.

1.3 Reading and Writing Files.

1.3.1 Getting Prepared To Use Files.

In each of the previous examples the user has been required to re-enter the student data - a somewhat tiring and boring situation. It would be preferable if the data could be entered once and then saved away for future use. (We have already done something similar when we saved our programs away for future use.) A collection of records, in this case the student records, is referred to as a *file*. This example shows how a file is defined. The next example uses the file.

```
*
* Define a File to Hold the Student Records.
*
 identification division.
 program-id. EXAMPLE-10.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'students'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 working-storage section.

 01   student-data.
      02 student-no              pic xxxx.
      02 name                    pic xxxxx.
      02 age                     pic xx.
      02 sex                     pic x.
```

```
procedure division.
    stop run.
```

Sample Program Execution

```
run
Execution begins...
...Execution ends.
```

Notes

(1) COBOL requires certain information in order to handle a file. A number of new statements are introduced in this example in order to define a file.

(2) The lines

```
input-output section.
file-control.
        select student-file
            assign to 'students'.
```

are placed in the environment division. They specify the *file-name* used by the program, "student-file", as well as the *system-name* of the file, "students".

(3) The data division has been modified to include the lines:

```
file section.
fd  student-file
    label records are standard.
01  student-record.
    02 pic x(60).
```

This section is used to describe some of the attributes of a particular file. It is also used to set-up an intermediate area into which data will be read or from which data will be written.

(4) The file definition, **fd,** defines the name of the file, namely "student-file". It is also necessary to tell the system how to deal with the *label* of the file. In this example, we indicate that labels will handled in a *standard* fashion. The reference manual expands on the concept of labels.

(5) The lines

```
01 student-record.
   02 pic x(60).
```

define a *record-name* for the file namely, "student-record" and specify that it contains 60 characters. The purpose of this area is to act as an *input-output* area to receive data from from the file or to send data to the file.

(6) When the record is read or written, 60 characters of information will be transmitted.

(7) Record-names and file-names are formed in the same way as data-names. System-names also consist of characters chosen from the letters and digits. The number of characters varies from system to system. Use of "short" system-names is usually safer, especially if one wants to use programs on a variety of systems.

Hint Since the student file will be used in many of the future examples, it is suggested that "texta" be modified to include the input-output and file section statements introduced in this example.

1.3.2 Create a Simple File.

Having defined the file in the previous example, we now accept student data and write it into the newly defined file.

```
*
* Write Student Records into a File.
*
 identification division.
 program-id. EXAMPLE-11.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
        select student-file
             assign to 'students'.

 data division.

 file section.
 fd    student-file
       label records are standard.
 01    student-record.
       02 filler         pic x(60).

 working-storage section.

 01    student-data.
       02 student-no            pic xxxx.
       02 name                  pic xxxxx.
       02 age                   pic xx.
       02 sex                   pic x.
```

```
procedure division.
     open output student-file.
     perform get-id.
     perform process-student-data
          until student-no = '9999'.
     move '9999' to student-no.
     write student-record from student-data.
     close student-file.
     stop run.

process-student-data.
     perform get-name.
     perform get-age.
     perform get-sex.
     write student-record from student-data.
     perform get-id.

get-id.
     display 'Enter student number (9999 to stop)'.
     move spaces to student-no.
     accept student-no.

get-name.
     display 'Enter name'.
     move spaces to name.
     accept name.

get-age.
     display 'Enter age'.
     move spaces to age.
     accept age.

get-sex.
     display 'Enter sex (M or F)'.
     accept sex.
```

Sample Program Execution

```
run
Execution begins...
Enter student number (9999 to stop)
4326
Enter name
Doug
Enter age
14
Enter sex (M or F)
M
Enter student number (9999 to stop)
3758
Enter name
Jane
Enter age
16
Enter sex (M or F)
F
Enter student number (9999 to stop)
6420
Enter name
Pat
Enter age
16
Enter sex (M or F)
F
Enter student number (9999 to stop)
9999
...Execution ends.
```

Notes

(1) The previous example defined the required file. This example will accept data as before and write it into the file.

(2) A file must be *opened* before it can be used. The statement

```
open output student-file
```

in the main paragraph of the procedure division specifies that the file is to be made available for output purposes.

(3) The program again requests the user to enter student number, name, age, and sex.

(4) The **display** verb cannot be used to write the data to the file. The statement

```
write student-record from student-data
```

causes the record to be written to the file. In effect this statementis saying, "Write the record as defined in the file definition and obtain the data from the area called student-data in working-storage." In fact, the data in working storage is moved to the file section and then it is written to the file.

(5) This process continues until the student number 9999 is entered.

(6) Future examples will want to read the file that has been created. In order to do this some means has to included in the file to recognize that we have read the last record i.e. we are at the end of the file. In our previous examples, entering 9999 accomplished this.

(7) A special *end-of-file* or *sentinel* record containing the student number 9999 is written after the last student record is accepted from the terminal.

(8) The statement

```
close student-file
```

releases the file indicating that the program no longer needs the file. It is not valid to specify "output" in a **close** sentence.

(9) The description of **open** and **close** is somewhat vague and omits many details about these two verbs. The description of the complete actions of **open** and **close** are described in the reference manual.

1.3.3 Read and Print a File.

In the previous example we created a student file. It would seem appropriate that we read this file and display the records to assure ourselves that they were written correctly.

```
*
* Read and Print the Student Records.
*
 identification division.
 program-id. EXAMPLE-12.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
         assign to 'students'.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler          pic x(60).

 working-storage section.

 01  student-data.
     02 student-no               pic xxxx.
     02 name                     pic xxxxx.
     02 age                      pic xx.
     02 sex                      pic x.
```

```
procedure division.
    open input student-file.
    perform read-student-record.
    perform process-student-data
        until student-no = '9999'.
    close student-file.
    stop run.

process-student-data.
    display student-data.
    perform read-student-record.

read-student-record.
    read student-file into student-data.
```

Sample Program Execution

```
run
Execution begins...
4326Doug    14M
3758Jane    16F
6420Pat     16F
...Execution ends.
```

Notes

(1) This program reads the file created in the previous example and displays each record as it appears in the file.

(2) The statement

```
open input student-file
```

specifies that the "student-file" is to be made available for input.

(3) The **accept** verb cannot be used to read a file. The statement

```
read student-file into student-data
```

causes a record to be read from the file and to be placed in working storage in the area called "student-data". In fact, the data is read into the file section and placed in the area called "student-record". It is then moved to working storage.

(4) Each record read is displayed exactly as it is contained in the file i.e. spaces may not be present between fields of the displayed record.

(5) When the last record containing a student number of 9999 is read, the reading and displaying process terminates.

(6) The **close** sentence releases the file.

(7) The file used in this and the previous example is referred to as a *sequential file*. Writing a sequential file means that each record is written immediately following the previous record. Reading a sequential file means that after a record has been read, the next record is then available to be read.

1.3.4 A Standard Method for Handling End of File.

In the example in which the file was created, we had to go to extra effort to create the sentinel record. Recognition of the end of a file is a common problem in file processing, and it should be no surprise that there is a standard method of dealing with the problem. If this were not the case, each program would require different and special tests to determine when the end of file was reached.

```
*
* At  End Clause.
*
 identification division.
 program-id. EXAMPLE-13.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
         assign to 'students'.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler        pic x(60).

 working-storage section.

 01  student-data.
     02 student-no              pic xxxx.
     02 name                    pic xxxxx.
     02 age                     pic xx.
     02 sex                     pic x.
```

```
procedure division.
     open input student-file.
     perform read-student-record.
     perform process-student-data
          until student-no = '9999'.
     close student-file.
     stop run.

process-student-data.
     display student-data.
     perform read-student-record.

read-student-record.
     read student-file into student-data
          at end move '9999' to student-no.
```

Sample Program Execution

```
run
Execution begins...
4326Doug    14M
3758Jane    16F
6420Pat     16F
...Execution ends.
```

Notes

(1) As a matter of course, whenever a file is created, a special end of file record is written. When examining a file, this record does not appear to be there as it is never displayed. The end-of-file record is automatically written when the file being created is closed using the **close** verb.

(2) This special record is recognized automatically whenever it is read by the system. This is accomplished by adding an extra clause in the **read** sentence which causes special processing when the end-of-file is read.

(3) When the end-of-file is encountered, no record is transferred to working storage. However, the **at end** clause is executed and in this case the constant 9999 is moved to the "student-no" field. This has same effect as reading the dummy or sentinel record. Recall that the **perform-until** checks this field to determine if there are any more records.

(4) The **at end** clause is executed only when the end-of-file record is read.

1.3.5 At End and High-values.

Using 9999 as the signal for an end-for-file might cause some potential problems. For example, someone might inadvertently assign a student the number 9999.

```
*
* At  End and High-Values.
*
 identification division.
 program-id. EXAMPLE-14.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
         assign to 'students'.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler          pic x(60).

 working-storage section.

 01  student-data.
     02 student-no              pic xxxx.
     02 name                    pic xxxxx.
     02 age                     pic xx.
     02 sex                     pic x.
```

```
procedure division.
    open input student-file.
    perform read-student-record.
    perform process-student-data
        until student-no = high-values.
    close student-file.
    stop run.

process-student-data.
    display student-data.
    perform read-student-record.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
4326Doug    14M
3758Jane    16F
6420Pat     16F
9999Pat     16F
...Execution ends.
```

Notes

(1) COBOL has a special constant known as **high-values** which can be used instead of 9999. In mathematical terms this quantity can be compared to *infinity*, in that no larger quantity can be assigned to the field.

(2) Both the **perform-until** and the **at end** clauses have been modified to use **high-values.**

(3) When the program is run an extra line is printed, namely the sentinel record with 9999 as the student number. Recall that this record was inserted by the program that created the file. Note also that the name, age, and sex are the same as the previous record. Recall that we only changed the student number before we wrote the sentinel record. This can be remedied by modifying the program that created the file.

(4) COBOL also has a constant called **low-values** which is the smallest possible value.

(5) **High-values** and **low-values** are called *figurative constants*. **Spaces** and **zero** are also figurative constants.

1.3.6 Use the File Provided with the System.

To save time and to provide some consistency, a version of the student file is provided with the system. This section describes the file and displays an unspaced listing of the file.

```
*
* Introduce the Student File.
*
 identification division.
 program-id. EXAMPLE-15.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
         assign to 'textfile'.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler        pic x(60).

 working-storage section.

 01  student-data.
     02 student-no          pic xxxx.
     02 name                pic x(20).
     02 age                 pic xx.
     02 sex                 pic x.
     02 class               pic x.
     02 school              pic x.
     02 algebra             pic xxx.
     02 geometry            pic xxx.
     02 physics             pic xxx.
     02 chemistry           pic xxx.
     02 english             pic xxx.
```

```
procedure division.
    open input student-file.
    perform read-student-record.
    perform process-student-data
        until student-no = high-values.
    close student-file.
    stop run.

process-student-data.
    display student-data.
    perform read-student-record.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution Begins...
1234Smith            SA  14m13075100075065084
1236Jones            TO  14m22076078055057078
1238Winterbourne     MS  14m31078088056067088
1239Harrison         K   14m42022087065087068
1240Graham           JW  14m21000068075067087
1242Welch            JW  14m31075075076075075
1243Dirksen          PH  14m42074085054068084
1245Cowan            DD  15f 33055066077088099
1249Sullivan         J   15f 42044055066077088
1256Kitchen          MP  14m43074049100097036
1266Taylor           YO  13f 33095083072066055
1268Allen            TT  13f 21098084073065059
1270Xerxes           X   13f 13099088077066055
1272Zimmerman        AB  13f 32095085078061057
1375Quantas          FL  15m22066066066066066
1388Beatle           RA  15f 11065062073076087
1390Cruikshank       TR  15f 33055064077076085
1393Hopper           BU  15f 23045069037026035
...Execution ends.
```

Notes

(1) In order to make the file available, the user is requested to run the program called "CBL43". A copy of this program appears as the last example in the tutorial section of the text.

(2) The student file contains 18 records.

(3) Each record in the file is 60 characters long and is composed of the following fields:

Student Number
The student number field is 4 digits in length and contains a 4-digit number.

Name
The name field is 20 characters in length and contains both surname and initials. The surname occupies the first 17 positions and the initials occupy the last 3 positions of this field.

Age
The age field is 2 characters in length and contains numbers in the range 12 to 19.

Sex
The sex field is 1 character in length and contains either an M or an F.

Class
The class field is 1 character in length and contains numbers in the range 1 to 4.

School
The school field is 1 character in length and contains numbers in the range 1 to 3.

Algebra, Geometry, Physics, Chemistry, and English
These five fields are all 3 digits in length and contain marks or grades for each of the subjects. Possible grades range from 0 to 100.

Space Reserved for Future Use.
The student record is defined to have 60 characters. The above fields occupy 44 characters of the record; the remaining 16 characters are reserved for future use.

(4) The system-name for the student file is "textfile".

(5) If the user plans to use the student file, it is suggested that the definition for "student-data" be entered as a **get** file.

1.3.7 Print a Report Using the Student File.

In all the examples presented to this point, we have used the **display** verb for producing output to the screen. It is more traditional in COBOL to define a special file for screen output and to use the **write** verb for displaying records.

```
*
* Print a Report Using the Student File.
*
 identification division.
 program-id. EXAMPLE-16.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
        select student-file
             assign to 'textfile'.
        select screen
             assign to 'terminal'.

 data division.

 file section.
 fd     student-file
        label records are standard.
 01     student-record.
        02 filler          pic x(60).

 fd     screen
        label records are standard.
 01     display-record.
        02 filler          pic x(80).
```

```
working-storage section.

01   student-data.
     02 student-no          pic xxxx.
     02 name.
        03surname           pic x(17).
        03initials          pic xxx.
     02 age                 pic xx.
     02 sex                 pic x.
     02 class               pic x.
     02 school              pic x.
     02 algebra             pic xxx.
     02 geometry            pic xxx.
     02 physics             pic xxx.
     02 chemistry           pic xxx.
     02 english             pic xxx.

01   report-heading.
     02 filler          pic x(20) value is spaces.
     02 filler          pic x(20) value is 'Student Reports'.
     02 filler          pic x(40) value is spaces.

01   first-line.
     02 out-student-no      pic x(4).
     02 filler              pic x(10) value is spaces.
     02 out-initials        pic xxx.
     02 filler              pic x value is spaces.
     02 out-surname         pic x(17).

01   second-line.
     02 filler              pic x(5) value is ' alg'.
     02 out-algebra         pic xxx.
     02 filler              pic x(5) value is ' gmt'.
     02 out-geometry        pic xxx.
     02 filler              pic x(5) value is ' phy'.
     02 out-physics         pic xxx.
     02 filler              pic x(5) value is ' chm'.
     02 out-chemistry       pic xxx.
     02 filler              pic x(5) value is ' eng'.
     02 out-english         pic xxx.

01   blank-line             pic x(80) value is spaces.
```

```
procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     perform process-student-data
          until student-no = high-values.
     close student-file
            screen.
     stop run.

process-student-data.
     perform display-student-data.
     perform read-student-record.

display-student-data.
     move student-no to out-student-no.
     move initials to out-initials.
     move surname to out-surname.
     write display-record from first-line.
     move algebra to out-algebra.
     move geometry to out-geometry.
     move physics to out-physics.
     move chemistry to out-chemistry.
     move english to out-english.
     write display-record from second-line.
     write display-record from blank-line.

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                                   Student Reports

1234          SA  Smith
 alg 075 gmt 100 phy 075 chm 065 eng 084

1236          TO  Jones
 alg 076 gmt 078 phy 055 chm 057 eng 078

1238          MS  Winterbourne
 alg 078 gmt 088 phy 056 chm 067 eng 088

1239          K   Harrison
 alg 022 gmt 087 phy 065 chm 087 eng 068

1240          JW  Graham
 alg 000 gmt 068 phy 075 chm 067 eng 087

1242          JW  Welch
 alg 075 gmt 075 phy 076 chm 075 eng 075

1243          PH  Dirksen
 alg 074 gmt 085 phy 054 chm 068 eng 084

1245          DD  Cowan
 alg 055 gmt 066 phy 077 chm 088 eng 099

1249          J   Sullivan
 alg 044 gmt 055 phy 066 chm 077 eng 088

1256          MP  Kitchen
 alg 074 gmt 049 phy 100 chm 097 eng 036

1266          YO  Taylor
 alg 095 gmt 083 phy 072 chm 066 eng 055

1268          TT  Allen
 alg 098 gmt 084 phy 073 chm 065 eng 059

1270          X   Xerxes
```

```
 alg 099 gmt 088 phy 077 chm 066 eng 055

1272         AB  Zimmerman
 alg 095 gmt 085 phy 078 chm 061 eng 057

1375         FL  Quantas
 alg 066 gmt 066 phy 066 chm 066 eng 066

1388         RA  Beatle
 alg 065 gmt 062 phy 073 chm 076 eng 087

1390         TR  Cruikshank
 alg 055 gmt 064 phy 077 chm 076 eng 085

1393         BU  Hopper
 alg 045 gmt 069 phy 037 chm 026 eng 035

...Execution ends.
```

Notes

(1) This example displays a report using the student file described in the previous example. The report consists of a heading followed by two lines for each student containing selected fields of the file. A blank line is displayed between each student report.

(2) While use of the **display** verb is appropriate to display records, it is more traditional to use the **write** verb. This, of course, requires the proper file definition. The file-name is called "screen" and the record-name defined in the file section is called "display-record". An area of 80 characters is defined since most screens can contain an 80 character line. However, it should be noted that some systems because of their hardware design will cause an additional line containing blanks to be printed if the 80th character is not a blank.

(3) The file "screen" is opened and closed in the appropriate places in the program.

(4) Records are displayed using the **write** verb in the form

```
write display-record from ...
```

(5) A record containing spaces is defined in working storage and is used to display blank lines.

(6) The name field of the student record consists of a 17 character surname followed by 3 characters for the initials. In this example we wish to display the initials before the surname. In order to do this we must define two fields for the name.

(7) COBOL permits us to subdivide a field further by introducing new level numbers and by using new data-names. The field "name" is divided into two fields "surname" and "initials" as follows:

```
02 name.
    03 surname    pic x(17).
    03 initials   pic xxx.
```

The **picture** clause has been removed from the 02-level item and two new items are defined at the 03-level; these new items are elementary items. Now the name can be referred to by using "name" or by using "surname" and/or "initials".

(8) The 03-level number items have been indented to improve readability.

(9) COBOL permits us to use up to 49 levels; the reference manual describes the rules of how these can be used.

1.3.8 Inputting a File Name.

We often want to change the system-name in a program. For example instead of directing output to the terminal, we might wish to display it on the printer. This example shows how this could be accomplished under program control.

```
*
* Input a File Name.
*
 identification division.
 program-id. EXAMPLE-17.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to ''.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard
      value of '' is output-file-name.
 01   display-record.
      02 filler          pic x(80).

 working-storage section.

 01   output-file-name             pic x(12).
```

```
01  student-data.
    02 student-no          pic xxxx.
    02 name                pic x(20).
    02 age                 pic xx.
    02 sex                 pic x.
    02 class               pic x.
    02 school              pic x.
    02 algebra             pic xxx.
    02 geometry            pic xxx.
    02 physics             pic xxx.
    02 chemistry           pic xxx.
    02 english             pic xxx.

01  report-heading.
    02 filler          pic x(20) value is spaces.
    02 filler          pic x(20) value is 'Student Reports'.
    02 filler          pic x(40) value is spaces.

01  first-line.
    02 out-student-no      pic x(4).
    02 filler              pic x(10) value is spaces.
    02 out-name            pic x(20).

01  second-line.
    02 filler              pic x(5) value is ' alg'.
    02 out-algebra         pic xxx.
    02 filler              pic x(5) value is ' gmt'.
    02 out-geometry        pic xxx.
    02 filler              pic x(5) value is ' phy'.
    02 out-physics         pic xxx.
    02 filler              pic x(5) value is ' chm'.
    02 out-chemistry       pic xxx.
    02 filler              pic x(5) value is ' eng'.
    02 out-english         pic xxx.

01  blank-line             pic x(80) value is spaces.
```

```
procedure division.
    display 'enter output file name - terminal or printer'.
    move spaces to output-file-name.
    accept output-file-name.
    open input student-file.
    open output screen.
    write display-record from report-heading.
    write display-record from blank-line.
    perform read-student-record.
    perform process-student-data
        until student-no = high-values.
    close student-file
          screen.
    stop run.

process-student-data.
    perform display-student-data.
    perform read-student-record.

display-student-data.
    move student-no to out-student-no.
    move name to out-name.
    write display-record from first-line.
    move algebra to out-algebra.
    move geometry to out-geometry.
    move physics to out-physics.
    move chemistry to out-chemistry.
    move english to out-english.
    write display-record from second-line.
    write display-record from blank-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
enter output file name - terminal or printer
printer
                         Student Reports

234        Smith          SA
alg 075 gmt 100 phy 075 chm 065 eng 084

236        Jones          TO
alg 076 gmt 078 phy 055 chm 057 eng 078

238        Winterbourne   MS
alg 078 gmt 088 phy 056 chm 067 eng 088

239        Harrison       K
alg 022 gmt 087 phy 065 chm 087 eng 068

240        Graham          JW
alg 000 gmt 068 phy 075 chm 067 eng 087

242        Welch           JW
alg 075 gmt 075 phy 076 chm 075 eng 075

243        Dirksen        PH
alg 074 gmt 085 phy 054 chm 068 eng 084

245        Cowan           DD
alg 055 gmt 066 phy 077 chm 088 eng 099

249        Sullivan       J
alg 044 gmt 055 phy 066 chm 077 eng 088

256        Kitchen        MP
alg 074 gmt 049 phy 100 chm 097 eng 036

266        Taylor         YO
alg 095 gmt 083 phy 072 chm 066 eng 055

268        Allen          TT
alg 098 gmt 084 phy 073 chm 065 eng 059
```

```
270         Xerxes         X
alg 099 gmt 088 phy 077 chm 066 eng 055

272         Zimmerman      AB
alg 095 gmt 085 phy 078 chm 061 eng 057

375         Quantas        FL
alg 066 gmt 066 phy 066 chm 066 eng 066

388         Beatle         RA
alg 065 gmt 062 phy 073 chm 076 eng 087

390         Cruikshank     TR
alg 055 gmt 064 phy 077 chm 076 eng 085

393         Hopper         BU
alg 045 gmt 069 phy 037 chm 026 eng 035

...Execution ends.
```

Notes

(1) This example prompts the user to enter the system-name, either "terminal" or "printer", and then displays a report similar to the previous example on the chosen device.

(2) Before running this example, the user should check local installation rules of printing. These will vary from system to system and from location to location.

(3) The **assign** clause for the "screen" has been changed to

```
assign screen to ''
```

i.e. the system-name has been made a null string.

(4) A new clause has been added to the **fd** statement, namely

```
value of '' is output-file-name.
```

where "output-file-name" is a data-name which is defined in working storage. The period has been placed following the new clause.

(5) Before the file is opened, the user is prompted to enter the proper output system-name, either "printer" or "terminal". If "terminal" is entered, the program functions exactly the same as the previous example. However, if "printer" is entered, the output will appear on the line-printer.

(6) On examining the output produced on the printer, the user may be somewhat surprised at the results. The style and format of the output will depend on the type of system and printer that are used. One possible result will be that the first character of each line will not be printed. A second result may be that the vertical spacing of the output seems somewhat bizarre. The next example will try to correct these unusual results.

1.3.9 Printer Control Characters.

In the previous example, the first character of each line was not displayed when the output was directed to the printer. Engineers have designed many printers so that the first print position is a code which provides the printer with information about vertical spacing. This position is not printed and must be supplied by the programmer. This special character is often referred to as the *print-control-character*.

```
*
* Print a Report on the Printer.
*
 identification division.
 program-id. EXAMPLE-18.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
         assign to 'textfile'.
     select screen
         assign to ''.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler         pic x(60).

 fd  screen
     label records are standard
     value of '' is output-file-name.
 01  display-record.
     02 filler         pic x(80).

 working-storage section.

 01  output-file-name          pic x(12).
```

```
01   student-data.
     02 student-no          pic xxxx.
     02 name                pic x(20).
     02 age                 pic xx.
     02 sex                 pic x.
     02 class               pic x.
     02 school              pic x.
     02 algebra             pic xxx.
     02 geometry            pic xxx.
     02 physics             pic xxx.
     02 chemistry           pic xxx.
     02 english             pic xxx.

01   report-heading.
     02 filler          pic x value is spaces.
     02 filler          pic x(20) value is spaces.
     02 filler          pic x(20) value is 'Student Reports'.
     02 filler          pic x(39) value is spaces.

01   first-line.
     02 filler              pic x     value is spaces.
     02 out-student-no      pic x(4).
     02 filler              pic x(10) value is spaces.
     02 out-name            pic x(20).

01   second-line.
     02 filler              pic x    value is spaces.
     02 filler              pic x(5) value is ' alg'.
     02 out-algebra         pic xxx.
     02 filler              pic x(5) value is ' gmt'.
     02 out-geometry        pic xxx.
     02 filler              pic x(5) value is ' phy'.
     02 out-physics         pic xxx.
     02 filler              pic x(5) value is ' chm'.
     02 out-chemistry       pic xxx.
     02 filler              pic x(5) value is ' eng'.
     02 out-english         pic xxx.

01   blank-line             pic x(80) value is spaces.
```

```
procedure division.
     display 'enter output file name - terminal or printer'.
     move spaces to output-file-name.
     accept output-file-name.
     open input student-file.
     open output screen.
     write display-record from report-heading
          after advancing 1 lines.
     write display-record from blank-line
          after advancing 1 lines.
     perform read-student-record.
     perform process-student-data
          until student-no = high-values.
     close student-file
            screen.
     stop run.

process-student-data.
     perform display-student-data.
     perform read-student-record.

display-student-data.
     move student-no to out-student-no.
     move name to out-name.
     write display-record from first-line
          after advancing 1 lines.
     move algebra to out-algebra.
     move geometry to out-geometry.
     move physics to out-physics.
     move chemistry to out-chemistry.
     move english to out-english.
     write display-record from second-line
          after advancing 1 lines
     write display-record from blank-line
          after advancing 1 lines.

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
enter output file name - terminal or printer
printer
                              Student Reports

1234        Smith          SA
 alg 075 gmt 100 phy 075 chm 065 eng 084

1236        Jones          TO
 alg 076 gmt 078 phy 055 chm 057 eng 078

1238        Winterbourne    MS
 alg 078 gmt 088 phy 056 chm 067 eng 088

1239        Harrison       K
 alg 022 gmt 087 phy 065 chm 087 eng 068

1240        Graham          JW
 alg 000 gmt 068 phy 075 chm 067 eng 087

1242        Welch          JW
 alg 075 gmt 075 phy 076 chm 075 eng 075

1243        Dirksen        PH
 alg 074 gmt 085 phy 054 chm 068 eng 084

1245        Cowan           DD
 alg 055 gmt 066 phy 077 chm 088 eng 099

1249        Sullivan       J
 alg 044 gmt 055 phy 066 chm 077 eng 088

1256        Kitchen        MP
 alg 074 gmt 049 phy 100 chm 097 eng 036

1266        Taylor         YO
 alg 095 gmt 083 phy 072 chm 066 eng 055

1268        Allen          TT
 alg 098 gmt 084 phy 073 chm 065 eng 059
```

```
1270        Xerxes         X
 alg 099 gmt 088 phy 077 chm 066 eng 055

1272        Zimmerman      AB
 alg 095 gmt 085 phy 078 chm 061 eng 057

1375        Quantas        FL
 alg 066 gmt 066 phy 066 chm 066 eng 066

1388        Beatle         RA
 alg 065 gmt 062 phy 073 chm 076 eng 087

1390        Cruikshank     TR
 alg 055 gmt 064 phy 077 chm 076 eng 085

1393        Hopper         BU
 alg 045 gmt 069 phy 037 chm 026 eng 035

...Execution ends.
```

Notes

(1) COBOL provides us with a number of ways of handling the print-control-character. This example shows one method; others are described in the reference manual.

(2) Another field is added to each of the record definitions for lines to be displayed. In each case, a field of one character is defined as the first character in the record. It is initialized to contain the blank character.

(3) Now when the program is run, the report is printed with "proper" spacing and containing all the desired characters.

(4) The use of the "blank" as the print-control-character indicates that we wish to do "single" spacing. Other characters are used for "double" and "triple" spacing. Another character is used to space the printer to the top of the page. These characters depend on the type of system and printer being used and are described in the reference manual.

(5) When this program is run with output directed to the screen, a blank character may be displayed at the start of each line. This means that for some screens, we will be limited to 79 characters.

(6) If the user plans to use both the terminal and the printer interchangeably in programs, it will be wise to plan for the use of the print-control-character. The remainder of the examples in this tutorial deal with output directed to the terminal.

1.4 Selection.

1.4.1 Selection Using the If Verb.

In previous examples which used the student file, we always displayed a line for each record in the file. Clearly on occasion we wish to display a sub-set or selection of lines from the file. The **if** verb gives us a way to make decisions in our COBOL programs and in particular to select and display certain records.

```
*
* If    Sentence.
*
 identification division.
 program-id. EXAMPLE-19.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).
```

```
working-storage section.

01  student-data.
    02 student-no          pic xxxx.
    02 name                pic x(20).
    02 age                 pic xx.
    02 sex                 pic x.
    02 class               pic x.
    02 school              pic x.
    02 algebra             pic xxx.
    02 geometry            pic xxx.
    02 physics             pic xxx.
    02 chemistry           pic xxx.
    02 english             pic xxx.

01  report-heading.
    02 filler          pic x(20) value is spaces.
    02 filler          pic x(25) value is 'Algebra Report'.
    02 filler          pic x(40) value is spaces.

01  display-line.
    02 out-student-no      pic x(4).
    02 filler              pic x(10) value is spaces.
    02 out-name            pic x(20).
    02 filler              pic x(5) value is spaces.
    02 out-algebra         pic xxx.

01  blank-line             pic x(80) value is spaces.

procedure division.
    open input student-file.
    open output screen.
    write display-record from report-heading.
    write display-record from blank-line.
    perform read-student-record.
    perform process-student-data
        until student-no = high-values.
    close student-file
          screen.
    stop run.
```

```
process-student-data.
     if algebra > '049'
          perform display-student-data.
     perform read-student-record.

display-student-data.
     move student-no to out-student-no.
     move name to out-name.
     move algebra to out-algebra.
     write display-record from display-line.

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                         Algebra Report

1234        Smith                    SA        075
1236        Jones                    TO        076
1238        Winterbourne             MS        078
1242        Welch                    JW        075
1243        Dirksen                  PH        074
1245        Cowan                    DD        055
1256        Kitchen                  MP        074
1266        Taylor                   YO        095
1268        Allen                    TT        098
1270        Xerxes                   X         099
1272        Zimmerman                AB        095
1375        Quantas                  FL        066
1388        Beatle                   RA        065
1390        Cruikshank               TR        055
...Execution ends.
```

Notes

(1) This example produces a report containing the student number, name, and algebra mark of those students whose algebra mark is 50 or greater.

(2) This can be accomplished by changing the "process-student-data" paragraph as follows:

```
process-student-data.
    if algebra > '049'
        perform display-student-data.
    perform read-student-record.
```

Note that we have introduced a new COBOL verb, namely **if.** When the **if** sentence is encountered during execution, the condition

```
algebra > '049'
```

is evaluated. If the condition is *true*, the paragraph "display-student-data" is executed. If the condition is *false*, control proceeds to the next sentence, and another record is read.

(3) The **if** sentence always contains a condition. These conditions are similar to those used with the **perform-until** sentence.

(4) The symbols '>' and '=' used in conditions in this example are called *relational operators*. Actually there is a third one namely, '<'. The word **not** can be included with each of the three conditions as follows:

```
not =
not <
not >
```

giving a total of six relational operators. It is also possible to use **equal** or even **equals** instead of '='. A complete list of alternatives can be found in the reference manual.

(5) The **if** sentence ends with a period. It is important to note that there is only one period, and this period terminates the sentence. The importance of this will become evident in the next example.

(6) The reader might be tempted to omit the 0(zero) from the '049' portion of the condition and write it as '49' or even ' 49'. If the program were run, it would be unlikely that one would obtain the correct results. Thus, it is usually necessary to include the 0(zero) in the '049'. The reader is referred to the reference manual to determine the reason for this.

(7) The **perform** portion of the **if** is indented for easier readability.

1.4.2 Another Version of If.

It is sometimes convenient to group a number of COBOL statements as a single entity. This example demonstrates how this can be done.

```
*
* If   Sentence (Another way).
*
 identification division.
 program-id. EXAMPLE-20.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).
```

```
working-storage section.

01  student-data.
    02 student-no           pic xxxx.
    02 name                 pic x(20).
    02 age                  pic xx.
    02 sex                  pic x.
    02 class                pic x.
    02 school               pic x.
    02 algebra              pic xxx.
    02 geometry             pic xxx.
    02 physics              pic xxx.
    02 chemistry            pic xxx.
    02 english              pic xxx.

01  report-heading.
    02 filler          pic x(20) value is spaces.
    02 filler          pic x(25) value is 'Algebra Report'.
    02 filler          pic x(40) value is spaces.

01  display-line.
    02 out-student-no       pic x(4).
    02 filler               pic x(10) value is spaces.
    02 out-name             pic x(20).
    02 filler               pic x(5) value is spaces.
    02 out-algebra          pic xxx.

01  blank-line              pic x(80) value is spaces.

procedure division.
    open input student-file.
    open output screen.
    write display-record from report-heading.
    write display-record from blank-line.
    perform read-student-record.
    perform process-student-data
        until student-no = high-values.
    close student-file
          screen.
    stop run.
```

```
process-student-data.
    perform display-student-data.
    perform read-student-record.

display-student-data.
    if algebra > '049'
        move student-no to out-student-no
        move name to out-name
        move algebra to out-algebra
        write display-record from display-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                    Algebra Report

1234     Smith          SA   075
1236     Jones          TO   076
1238     Winterbourne   MS   078
1242     Welch          JW   075
1243     Dirksen        PH   074
1245     Cowan          DD   055
1256     Kitchen        MP   074
1266     Taylor         YO   095
1268     Allen          TT   098
1270     Xerxes         X    099
1272     Zimmerman      AB   095
1375     Quantas        FL   066
1388     Beatle         RA   065
1390     Cruikshank     TR   055
...Execution ends.
```

Notes

(1) This example is another version of the previous example. It presents the concept of the *range* of the **if** and shows how a series of COBOL statements can be executed when the condition is true.

(2) The **if** sentence has been moved to the "display-student-data" paragraph. The periods have been removed from each of the sentences except the last in this paragraph. The **if** is followed by the four statements and is terminated by a period to form the **if** sentence. The statements in the *range* of the **if** are executed if the condition is true.

(3) All statements in the range of the **if** are indented for readability.

(4) The importance of the period cannot be over-emphasized. It terminates the **if** and no other periods should be placed in the range of the **if.**

(5) The word *statement* has been and will be used to refer to a COBOL sentence without a period.

(6) While the versions of the **if** in this and the previous example both function properly, it is suggested that one avoid, if possible, "large" **if** sentences.

1.4.3 The Else Option.

In the previous two examples, we did not require a line to be displayed if the algebra mark was less than 50. It is sometimes the case that we wish to perform one action if a condition is true and an alternative action if the condition is false; for example, one action if the mark is less than 50 and another if the mark is greater or equal to 50.

```
*
* Else Option.
*
 identification division.
 program-id. EXAMPLE-21.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
         assign to 'textfile'.
     select screen
         assign to 'terminal'.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler          pic x(60).

 fd  screen
     label records are standard.
 01  display-record.
     02 filler          pic x(80).
```

```
working-storage section.

01   student-data.
     02  student-no            pic xxxx.
     02 name                   pic x(20).
     02  age                   pic xx.
     02  sex                   pic x.
     02  class                 pic x.
     02 school                 pic x.
     02  algebra               pic xxx.
     02  geometry              pic xxx.
     02  physics               pic xxx.
     02  chemistry             pic xxx.
     02  english               pic xxx.

01   report-heading.
     02 filler            pic x(20) value is spaces.
     02 filler            pic x(20) value is 'Pass - Fail Report'.
     02 filler            pic x(40) value is spaces.

01   display-line.
     02  out-student-no        pic x(4).
     02  filler                pic x(10) value is spaces.
     02  out-name              pic x(20).
     02  filler                pic x(5) value is spaces.
     02  out-algebra           pic xxx.
     02  filler                pic x(5) value is spaces.
     02  pass-fail             pic x(10).

01   blank-line                pic x(80) value is spaces.

procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     perform process-student-data
          until student-no = high-values.
     close student-file
            screen.
     stop run.
```

```
process-student-data.
    perform display-student-data.
    perform read-student-record.

display-student-data.
    if algebra < '050'
        move 'failed' to pass-fail
    else
        move 'passed' to pass-fail.
    move student-no to out-student-no.
    move name to out-name.
    move algebra to out-algebra.
    write display-record from display-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                    Pass - Fail Report

1234     Smith           SA    075    passed
1236     Jones           TO    076    passed
1238     Winterbourne    MS    078    passed
1239     Harrison        K     022    failed
1240     Graham          JW    000    failed
1242     Welch           JW    075    passed
1243     Dirksen         PH    074    passed
1245     Cowan           DD    055    passed
1249     Sullivan        J     044    failed
1256     Kitchen         MP    074    passed
1266     Taylor          YO    095    passed
1268     Allen           TT    098    passed
1270     Xerxes          X     099    passed
1272     Zimmerman       AB    095    passed
1375     Quantas         FL    066    passed
1388     Beatle          RA    065    passed
1390     Cruikshank      TR    055    passed
1393     Hopper          BU    045    failed
...Execution ends.
```

Notes

(1) In this example a report is produced displaying student number, name, and algebra mark for all students as well as a field indicating if the student passed or failed algebra.

(2) A new field has been included in "display-line" to contain the pass-fail indication.

(3) The "display-student-data" paragraph has been altered to include the sentence

```
if algebra < '050'
    move 'failed' to pass-fail
else
    move 'passed' to pass-fail.
```

(4) The condition is tested and if it is true, 'failed' is moved to the display record; if it is false, 'passed' is moved. We refer to the two actions as being contained in the *true range* and *false range* of the **if** sentence. The true range ends with the **else** and the false range ends with the period.

(5) The **else** is placed on a separate line and both the true and false ranges are indented for readability.

1.4.4 Multiple Choice.

In the previous example we caused either "passed" or "failed" to be displayed in the record. This can be thought of as two *cases*. However, many situations arise where more than two cases are involved.

```
*
* Multiple Choice.
*
 identification division.
 program-id. EXAMPLE-22.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
         assign to 'textfile'.
     select screen
         assign to 'terminal'.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler        pic x(60).

 fd  screen
     label records are standard.
 01  display-record.
     02 filler        pic x(80).
```

```
working-storage section.

01   student-data.
     02  student-no          pic xxxx.
     02  name                pic x(20).
     02  age                 pic xx.
     02  sex                 pic x.
     02  class               pic x.
     02  school              pic x.
     02  algebra             pic xxx.
     02  geometry            pic xxx.
     02  physics             pic xxx.
     02  chemistry           pic xxx.
     02  english             pic xxx.

01   report-heading.
     02  filler          pic x(20) value is spaces.
     02  filler          pic x(25) value is 'Letter - Grade Report'.
     02  filler          pic x(35) value is spaces.

01   first-line.
     02  out-student-no      pic x(4).
     02  filler              pic x(10) value is spaces.
     02  out-name            pic x(20).
     02  filler              pic x(5) value is spaces.
     02  out-algebra         pic xxx.
     02  filler              pic x(5) value is spaces.
     02  grade               pic x.

01   blank-line              pic x(80) value is spaces.

procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     perform process-student-data
          until student-no = high-values.
     close student-file
             screen.
     stop run.
```

```
process-student-data.
     perform display-student-data.
     perform read-student-record.

display-student-data.
     if      algebra < '050'
          move 'F' to grade
     else if algebra < '060'
          move 'D' to grade
     else if algebra < '066'
          move 'C' to grade
     else if algebra < '075'
          move 'B' to grade
     else
          move 'A' to grade.
     move student-no to out-student-no.
     move name to out-name.
     move algebra to out-algebra.
     write display-record from first-line.

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                              Letter - Grade Report

1234      Smith                SA     075    A
1236      Jones                TO     076    A
1238      Winterbourne         MS     078    A
1239      Harrison             K      022    F
1240      Graham               JW     000    F
1242      Welch                JW     075    A
1243      Dirksen              PH     074    B
1245      Cowan                DD     055    D
1249      Sullivan             J      044    F
1256      Kitchen              MP     074    B
1266      Taylor               YO     095    A
1268      Allen                TT     098    A
1270      Xerxes               X      099    A
1272      Zimmerman            AB     095    A
1375      Quantas              FL     066    B
1388      Beatle               RA     065    C
1390      Cruikshank           TR     055    D
1393      Hopper               BU     045    F
...Execution ends.
```

Notes

(1) The example produces a report which includes letter grades as well as the numeric values. The letter A, B, C, D, or F is displayed in the appropriate situation.

(2) The "display-student-data" paragraph now contains the more complicated **if** *which handles the necessary cases.*

```
if      algebra < '050'
        move 'F' to grade
else if algebra < '060'
        move 'D' to grade
else if algebra < '066'
        move 'C' to grade
else if algebra < '075'
        move 'B' to grade
else
        move 'A' to grade.
```

(3) Here we have a number of **if**'s with the entire **if** sentence ending with a single period. If the algebra mark is less than 50, an F is moved to the display line and then control passes to the next sentence. If the mark is not less than 50 control passes to the clause

```
else if algebra < '060'
```

Here the program asks is the mark is less than 60 and if that is the case, a D is moved to the display line and then control passes to the next sentence. If not control passes to the next **else if** clause. This process continues until the correct condition is found.

(4) The **else if** clause

```
else if algebra < '060'
```

determines if the mark is in the range 50 to 59 since the previous condition

```
algebra < '050'
```

eliminated the case of all marks less than 50. Similarly, at each of the other **else if** clauses, the program checks for the correct range of marks.

(5) The **if** sentence could have been written as follows:

```
if algebra < '050'
     move 'F' to grade
else
     if algebra < '060'
          move 'D' to grade
     else
          if algebra < '066'
               move 'C' to grade
          else
               if algebra < '075'
                    move 'B' to grade
               else
                    move 'A' to grade.
```

Either style of the **if** sentence is acceptable. However, the authors prefer the style used in the program.

1.4.5 Logical Operators - And and Or.

On many occasions we wish to test more than one condition. The *logical operators* **and** and **or** can be used to accomplish this.

```
*
* And and Or.
*
 identification division.
 program-id. EXAMPLE-23.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
         assign to 'textfile'.
     select screen
         assign to 'terminal'.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler        pic x(60).

 fd  screen
     label records are standard.
 01  display-record.
     02 filler        pic x(80).
```

```
working-storage section.

01   student-data.
     02  student-no            pic xxxx.
     02 name                   pic x(20).
     02  age                   pic xx.
     02  sex                   pic x.
     02  class                 pic x.
     02 school                 pic x.
     02  algebra               pic xxx.
     02  geometry              pic xxx.
     02  physics               pic xxx.
     02  chemistry             pic xxx.
     02  english               pic xxx.

01   report-heading.
     02  filler            pic x(20) value is spaces.
     02  filler            pic x(30)
           value is 'Class 2 - Algebra Report'.
     02  filler            pic x(40) value is spaces.

01   first-line.
     02  out-student-no        pic x(4).
     02  filler                pic x(10) value is spaces.
     02  out-name              pic x(20).
     02  filler                pic x(5) value is spaces.
     02 out-algebra            pic xxxxx.

01   blank-line                pic x(80) value is spaces.

procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     perform process-student-data
         until student-no = high-values.
     close student-file
           screen.
     stop run.
```

```
process-student-data.
    if algebra > '074' and class = '2'
        perform display-student-data.
    perform read-student-record.

display-student-data.
    move student-no to out-student-no.
    move name to out-name.
    move algebra to out-algebra.
    write display-record from first-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                        Class 2 - Algebra Report

1236      Jones                        TO       076
1268      Allen                        TT       098
...Execution ends.
```

Notes

(1) This example produces a report of students in the second class whose algebra mark is 75 or greater.

(2) In this case the **if** sentence

```
if algebra > '074' and class = '2'
    perform display-student-data.
```

uses a *compound condition* with the logical operator **and.** If both conditions are true the record is displayed. If either or both of the conditions are false, the record is not displayed.

(3) If the logical operator **or** were used instead of **and** in this example, records would be displayed for all students in the second class as well as all those in other classes whose algebra mark was 75 or greater.

1.4.6 Combined Use of And and Or

On occasion we wish to combine the logical operators **and** and **or** in a compound condition.

```
*
* Compound Conditions.
*
 identification division.
 program-id. EXAMPLE-24.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
         assign to 'textfile'.
     select screen
         assign to 'terminal'.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler          pic x(60).

 fd  screen
     label records are standard.
 01  display-record.
     02 filler          pic x(80).
```

```
working-storage section.

01   student-data.
     02  student-no           pic xxxx.
     02 name                  pic x(20).
     02  age                  pic xx.
     02  sex                  pic x.
     02  class                pic x.
     02 school                pic x.
     02  algebra              pic xxx.
     02  geometry             pic xxx.
     02  physics              pic xxx.
     02  chemistry            pic xxx.
     02  english              pic xxx.

01   report-heading.
     02 filler           pic x(20) value is spaces.
     02 filler           pic x(30)
          value is 'Class 2 & 4 - Algebra Report'.
     02 filler           pic x(40) value is spaces.

01   first-line.
     02  out-student-no       pic x(4).
     02  filler               pic x(10) value is spaces.
     02  out-name             pic x(20).
     02  filler               pic x(5) value is spaces.
     02  out-class            pic x.
     02  filler               pic xx value is spaces.
     02  out-algebra          pic xxxxx.

01   blank-line               pic x(80) value is spaces.

procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     perform process-student-data
          until student-no = high-values.
     close student-file
           screen.
     stop run.
```

```
process-student-data.
     if    (algebra > '074' and class = '2')
       or
            (algebra < '050' and class = '4')
          perform display-student-data.
     perform read-student-record.

display-student-data.
     move student-no to out-student-no.
     move name to out-name.
     move class to out-class.
     move algebra to out-algebra.
     write display-record from first-line.

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                        Class 2 & 4 - Algebra Report

1236      Jones                   TO      2   076
1239      Harrison                K       4   022
1249      Sullivan                J       4   044
1268      Allen                   TT      2   098
...Execution ends.
```

Notes

(1) This example produces a report for students in the second class whose algebra mark is 75 or greater as well as students in the fourth class whose algebra mark is less than 50.

(2) The compound condition

```
          (algebra > '074' and class = '2')
     or
          (algebra < '050' and class = '4')
```

performs the required test. It determines if the class 2 - algebra condition is true and then if the class 4- algebra condition is true. If either is true the record is displayed.

(3) Parentheses have been introduced in order that the conditions are evaluated in the desired order and to make the compound condition easier to understand. Quantities enclosed in parentheses are evaluated first.

(4) If parentheses are omitted, **and**'s are evaluated first, followed by **or**'s. Thus in this example the parentheses could have been omitted. However, they were included to reduce possible ambiguity.

(5) The following compound condition illustrates the use of parentheses.

```
    (class = '2' or class = '4')
and
    (algebra < '050' or algebra > '075')
```

In this case, the report would contain students in the second or fourth class who had marks less than 50 or greater than 75. If parentheses were omitted the report would contain all students in class 2, students in class 4 with algebra marks less than 50, and students whose algebra mark was greater than 75.

(6) Finally, this new compound condition could be written somewhat more compactly as

```
    (class = '2' or '4')
and
    (algebra > '075' or < '050')
```

The reader is referred to the reference manual for a more complete presentation of *implied subjects* in compound conditions.

1.5 Arithmetic.

1.5.1 Integer Arithmetic

One of the major uses of computers is to perform arithmetic. This example introduces the verbs used for arithmetic operations. It also presents a new field definition for defining numbers to be used in arithmetic operations. The program itself has little meaning; it is used to demonstrate arithmetic operations.

```
*
* Simple Arithmetic (Integer Numbers).
*
 identification division.
 program-id. EXAMPLE-25.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01   a      pic 9(4) value is 1234.
 01   b      pic 9(6) value is 123456.

 01   c      pic 9(7).
 01   d      pic 9(6).
 01   e      pic 9(10).
 01   f      pic 9(3).
```

```
procedure division.
    add a b giving c.
    subtract a from b giving d.
    multiply a by b giving e.
    divide a into b giving f.
    display 'a ' a.
    display 'b ' b.
    display ' '.
    display 'c ' c.
    display 'd ' d.
    display 'e ' e.
    display 'f ' f.
    stop run.
```

Sample Program Execution

```
run
Execution begins...
a  1234
b  123456

c  0124690
d  122222
e  0152344704
f  100
...Execution ends.
```

Notes

(1) This example defines two fields "a" and "b", assigns the values 1234 and 123456, and then finds the sum, difference, product, and quotient of the two values. The answers of the four operations are stored in four fields and are then displayed.

(2) A different **picture** clause is required for fields which are used to store numbers which will be used to enter into arithmetic operations. The two definitions for "a" and "b" are written as

```
01  a pic 9(4) value is 1234.
01  b pic 9(6) value is 123456.
```

Here "a" holds a 4-digit number with the value 1234 and "b" holds a 6-digit number with the value 123456. The two definitions could have been written as:

```
01  a pic 9999 value is 1234.
01  b pic 999999 value is 123456.
```

(3) It is a basic rule of COBOL that fields that are to enter into arithmetic operations *must* be declared using **picture**'s with 9's instead of x's. These new **picture**'s are referred to as *numeric pictures*.

(4) The integer values, 1234 and 123456, are not enclosed by quotation marks. They are referred to as *numeric literals*.

(5) The four basic sentences used in this example for doing arithmetic operations are:

```
add a b giving c.
subtract a from b giving d.
multiply a by b giving e.
divide a into b giving f.
```

The meaning of each of these sentences is fairly obvious. In each case the contents of the fields "a" and "b" enter into an arithmetic operation, and the answers are stored in "c", "d", "e", and "f" respectively.

(6) The fields "c", "d", "e", and "f" are the receiving fields for the four computations. When these fields have more positions than needed to hold the answer, zeros are padded on the left.

(7) When division is performed the result is stored in "f" giving a value 100. The remainder, namely 56, is lost. Later we will see how the remainder can be retained.

(8) A number of other forms of the arithmetic verbs exist in COBOL. These are described in the reference manual. Several are presented in future examples.

1.5.2 Decimal Places

In the previous example, the decimal point was assumed to be to the right of the right-most digit. This example introduces decimal values in arithmetic operations.

```
*
* Simple Arithmetic (With Decimal Places).
*
 identification division.
 program-id. EXAMPLE-26.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01   a      pic 99v99 value is 12.34.
 01   b      pic 999v999 value is 123.456.

 01   c      pic 9(7).
 01   d      pic 999v999.
 01   e      pic 9(5)v999.
 01   f      pic 999v999.

 procedure division.
     add a b giving c.
     subtract a from b giving d.
     multiply a by b giving e.
     divide a into b giving f.
     display 'a ' a.
     display 'b ' b.
     display ' '.
     display 'c ' c.
     display 'd ' d.
     display 'e ' e.
     display 'f ' f.
     stop run.
```

Sample Program Execution

```
run
Execution begins...
a  1234
b  123456

c  0000135
d  111116
e  01523447
f  010004
...Execution ends.
```

Notes

(1) To indicate a decimal point in the field, we use a 'v' in the appropriate place in the **picture** clause. For example,

```
01  a pic 99v99 value is 12.34.
01  b pic 999v999 value is 123.456.
```

defines a 4-digit field which has two decimal places, and a 6-digit field which has 3 decimal places. The values 12.34 and 123.456 are assigned to the two fields.

(2) Consider the statement

```
add a b giving c.
```

Here the computer lines up the decimal points and performs the addition as follows:

```
 12.34
123.456
-------
135.796
```

The result is stored in "c". However, "c" has a **picture** of 9(7) meaning it can hold a 7-digit integer. Hence, the portion of the result to the right of the decimal is dropped or truncated. Only the 135 is stored with four zeroes inserted on the left to fill out the field.

(3) In the case of the subtract operation, we were not satisfied with the truncation of the result to an integer. The **picture** clause for "d" is defined as

```
01  d pic 999v999.
```

and the appropriate value is stored in "d" namely 111116. Note that no decimal point is displayed i.e. we have to remember where it is. In a similar way, we have included decimal places for "e" and "f".

(4) There is no decimal point physically recorded in working storage. The symbol 'v' is used to indicate its position.

(5) Numeric literals can contain decimal points, as illustrated by the values 12.34 and 123.456.

(6) Consider the statement

```
add a b giving c rounded
```

and assume that the **picture** clause for "c" has been defined as

```
01  c  pic 99999v99.
```

Here the value displayed would be 0013580. Since the **picture** clause is defined to have 2 digits to the right of the decimal, the third digit after the decimal is examined and if it is five or greater, the result is *rounded*. In this example, the third digit after the decimal is a 6 and hence the value is rounded. The **rounded** option can be used with the other arithmetic verbs.

(7) Assume that the **picture** clause for "e" had been written as

```
01  e  pic 999v999.
```

In this case the result for "e" would have been 523447 i.e. the 1 has been dropped from the result. COBOL does *not* check if the result is too large for the receiving field; it merely truncates the result to fit in the receiving field. This can be remedied by using the **on size error** option. For example,

```
multiply a by b giving e
     on size error display 'arithmetic overflow'.
```

would tell the user that an error had occurred and an appropriate action could be taken. This option can also be used with all the arithmetic verbs.

(8) The reader is referred to the reference manual for a more complete description of **rounded** and **on size error.**

1.5.3 Negative Numbers

The previous two examples used only positive values. This example shows how negative values can be defined and used.

```
*
* Simple Arithmetic (With Decimal Places and Negative Numbers).
*
 identification division.
 program-id. EXAMPLE-27.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01  a     pic s99v99 value is -12.34.
 01  b     pic s999v999 value is 123.456.

 01  c     pic s9(4)v999.
 01  d     pic s9(4)v99.
 01  e     pic s9(6)v99999.
 01  f     pic s9(3)v999.

 procedure division.
     add a b giving c.
     subtract a from b giving d.
     multiply a by b giving e.
     divide a into b giving f.
     display 'a ' a.
     display 'b ' b.
     display ' '.
     display 'c ' c.
     display 'd ' d.
     display 'e ' e.
     display 'f ' f.
     stop run.
```

Sample Program Execution

```
run
Execution begins...
a  123M
b  12345F

c  011111F
d  01357I
e  0015234470M
f  01000M
...Execution ends.
```

Notes

(1) We wish to assign the value -12.34 to "a" rather than 12.34. This is done by changing the **picture** for "a" to

```
02 a pic s99v99 value is -12.34.
```

(2) The numeric literal is written as one would expect, namely -12.34.

(3) The **picture** has been changed to s99v99 from 99v99. The character 's' is included to indicate that the field may contain a negative value and that provision must be made to store a sign.

(4) The other **picture** clauses have also been changed in anticipation that the respective fields may contain negative values. Clearly the definition for "b" need not have been changed since "b" contains a positive value. However, without doing the actual arithmetic operations, we cannot be sure if "c", "d", "e", or "f" will contain positive or negative results.

(5) If the 's' is omitted and the result of an arithmetic operation is negative, the result will be stored as a positive value. Hence it is safer to include the 's' for all fields unless one is certain that the value to be stored is positive.

(6) Upon examination of the output, the user is probably somewhat disconcerted to find that the right-most character of each of the results may be a letter instead of a digit. In fact the output produced by your system may differ from that shown in this example. This depends on the particular coding or *collating sequence* used by your system. You should refer to the reference manual if your output differs.

(7) In COBOL the sign is stored as part of the right-most digit in the field. This results in unexpected letters being displayed. The letters A - I represent the positive integers 1 - 9 and the letters J - R represent the negative integers 1 - 9. Thus, 123M is -12.34 (the 'v' places the decimal) and 12345F is 123.456. Positive zero is represented by { and negative zero is }. A future example will show how to display negative values in a more appropriate fashion.

1.5.4 Expressions and the Compute Verb.

In the previous examples, we were limited to one arithmetic operation for each verb. Mathematicians have provided us with a language for expressing arithmetic computations, namely algebra. COBOL has a facility to incorporate certain aspects of this language by permitting one to define a mathematical expression and to then **compute** the required value.

```
*
* Compute Verb.
*
 identification division.
 program-id. EXAMPLE-28.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01  a     pic s99v99 value is -12.34.
 01  b     pic s999v999 value is 123.456.

 01  c     pic s999v999.
 01  d     pic s9(6)v9(6).
 01  e     pic s99v99.
 01  f     pic s9(3).

 procedure division.
     compute c = a + b.
     compute d = a + b * c + 425.2.
     compute e = b / a * (b - a).
     compute f = 4 ** 3.
     display 'a ' a.
     display 'b ' b.
     display ' '.
     display 'c ' c.
     display 'd ' d.
     display 'e ' e.
     display 'f ' f.
     stop run.
```

Sample Program Execution

```
run
Execution begins...
a  123M
b  12345F

c  11111F
d  01413079689F
e  585P
f  06D
...Execution ends.
```

Notes

(1) This example contains a number of arithmetic expressions listed below. A few others are included for completeness.

```
a + b
a + b * c + 425.2
b / a * (b - a)
4 ** 3
a
- a + b
7.9
b ** - 2
```

Each example consists of one or more numeric quantities, combined with the symbols +, -, *, /, and **, which represent addition, subtraction, multiplication, division, and exponentiation respectively. The numeric quantities are data-names representing numeric fields or numeric literals. Parentheses are used to denote operations which are to be evaluated first.

(2) Just as in algebra, priority rules are used to determine the order of computation of the items in an expression. Quantities in parentheses are considered sub-expressions and are evaluated first, inner-most parentheses receive priority over outer parentheses. The priority of operators is as follows, arranged in descending order.

-	unary minus
**	exponentiation
* /	multiply and divide
+ -	add and subtract

Whenever any ambiguity exists, such as in the third example, the left-most operation is performed first.

(3) The expressions are entered with a space on either side of the operator. If spaces were not the rule, it would be difficult to determine if a-b were a data-name or an expression.

(4) The **compute** verb in the statement

```
compute d = a + b * c + 452.2
```

causes the expression to be evaluated and its value is assigned to "d".

(5) Spaces must appear on either side of the equal sign.

(6) The data-name to the left of the equal sign may appear in the arithmetic expression. Consider the example

```
compute a = a + 1.
```

Here the expression "a + 1" is evaluated first. Then the result is assigned to "a".

(7) Expressions may also be used in conditions. For example in the condition

```
a + 4 > b - 234
```

both arithmetic expressions are evaluated and then the comparison is made.

1.6 Printing and Editing Numeric Values.

1.6.1 Decimals in Output.

The previous several examples introduced how one can do arithmetic operations. However, the output produced left much to be desired in that it contained high-order zeroes, no decimal points, and even displayed letters instead of digits in the case of signed values. This example shows how decimal points can be included as part of the output. Future examples will show various ways of *editing* output values to have a more reasonable appearence.

```
*
* Displaying the Decimal Point.
*
 identification division.
 program-id. EXAMPLE-29.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01  a    pic 99V99 value is 12.34.
 01  b    pic 999V999 value is 123.456.

 01  c    pic 9(5).99.
 01  d    pic 9(5).999.
 01  e    pic 9(7).99999.
 01  f    pic 9(3).999.
```

```
procedure division.
    add a b giving c.
    subtract a from b giving d.
    multiply a by b giving e.
    divide a into b giving f.
    display 'a ' a.
    display 'b ' b.
    display ' '.
    display 'c ' c.
    display 'd ' d.
    display 'e ' e.
    display 'f ' f.
    stop run.
```

Sample Program Execution

```
run
Execution begins...
a  1234
b  123456

c  00135.79
d  00111.116
e  0001523.44704
f  010.004
...Execution ends.
```

Notes

(1) The four lines of output for "c", "d", "e", and "f" now include the decimal point in the correct position. This is accomplished by changing the four **picture** clauses to contain a '.' instead of a 'v'.

(2) Because the decimal points are inserted, an extra character appears in each of the fields that are displayed.

(3) By writing '.' instead of 'v' the decimal point actually appears in working storage. (Recall that in the case of 'v' only the position was recorded and no physical decimal point was inserted.) Since the decimal point is present, the field is *not* a legitimate *numeric* field, and cannot therefore enter into arithmetic operations. Thus it would be incorrect to change the definition of "a" as follows:

```
01 a pic 99.99 value is 12.34.
```

(4) **Picture** clauses containing the '.' are called *output pictures* and the data stored in them are called *numeric edited* data. They cannot be used in arithmetic operations. They can be used only to receive results from arithmetic operations and are used solely for display purposes.

1.6.2 Suppress Leading Zeros and Printing Minus Signs.

Leading zeros are usually not considered necessary when displaying numeric results. This example shows how they can be eliminated.

```
*
* Suppress Leading Zeroes.
*
 identification division.
 program-id. EXAMPLE-30.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01  a    pic 99V99 value is 12.34.
 01  b    pic 999V999 value is 123.456.

 01  c    pic z(5).99.
 01  d    pic z(5).99.
 01  e    pic z(7).99999.
 01  f    pic z(3)9.999.

 procedure division.
     add a b giving c.
     subtract a from b giving d.
     multiply a by b giving e.
     divide a into b giving f.
     display 'a ' a.
     display 'b ' b.
     display ' '.
     display 'c ' c.
     display 'd ' d.
     display 'e ' e.
     display 'f ' f.
     stop run.
```

Sample Program Execution

```
run
Execution begins...
a  1234
b  123456

c      135.79
d      111.11
e       1523.44704
f      10.004
...Execution ends.
```

Notes

(1) The output no longer has leading zeroes. The symbol 'z' acts precisely like a '9' except that if the digit to be displayed is a '0', and no non-zero digit appears to the left of it, it is replaced by a blank character.

(2) It is common practice to have '9's' after the decimal place in output **pictures.** If "c" had been defined to have 2 z's after the decimal and the value to be displayed were 00000.00, all that would be displayed would be spaces. By using 9's we are assured that .00 would be displayed.

(3) If a value to be displayed is negative, we will wish to display a minus sign to the left of the value.

(4) The '-' character behaves as a 'z' with one extra feature. Should the number be negative, a minus sign is placed immediately to the left of the first non-blank character to be displayed. If the **picture** clauses for were written as

```
01  a   pic s99v99 value is -12.34.
01  b   pic 999v999 value is 123.456.

01  c   pic -(5).99.
01  d   pic -(5).99.
01  e   pic -(7).99999.
01  f   pic -(3).999.
```

and the output displayed would be

```
a  123M
b  123456

c  111.11
d  135.79
e  -1523.44704
f  -10.004
```

1.6.3 Dollar Signs, Commas, and CR.

On many occasions the values we wish to display represent some form of money, usually dollars. In this case we may wish to display the $ sign as well as inserting commas to make the output more readable. Finally, accountants rarely display a minus sign if a value is negative. They usually use the CR symbol. These features are incorporated in this example.

```
*
* Printing Dollar Signs, Commas, and CR.
*
 identification division.
 program-id. EXAMPLE-31.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01   a     pic s9999V99 value is -123.40.
 01   b     pic s999999V99 value is 12345.60.

 01   c     pic $(7).99.
 01   d     pic $$$,$$$,$$$.99CR.
 01   e     pic $$$,$$$,$$$.99CR.
 01   f     pic $$$,$$$,$$$.99CR.

 procedure division.
     add a b giving c.
     subtract a from b giving d.
     multiply a by b giving e.
     divide a into b giving f.
     display 'a ' a.
     display 'b ' b.
     display ' '.
     display 'c ' c.
     display 'd ' d.
     display 'e ' e.
     display 'f ' f.
     stop run.
```

Sample Program Execution

```
run
Execution begins...
a  01234}
b  0123456{

c    $12222.20
d          $12,469.00
e     $1,523,447.04CR
f             $100.04CR
...Execution ends.
```

Notes

(1) The '$' character also behaves like a 'z' with one extra feature. A '$' is inserted immediately to the left of the left-most non-blank character. Examine the value displayed for "c".

(2) In the last three lines of output, commas are inserted in the **picture** clauses. This causes commas to be displayed whenever there are significant digits to the left of it.

(3) The values to be displayed for "e" and "f" are negative and instead of displaying a minus sign to the left of the value, the symbol CR is displayed to the right of the value.

(4) Finally, note that as extra characters such as comma, CR, '.' etc. are included in the **picture** clauses, extra characters appear in the displayed values. Note also that when displaying "d" no CR is present but it should be noted that two blank spaces have been inserted.

1.6.4 Combining Edit Characters

In order to achieve the desired form of output, it is often necessary to combine some of the edit characters. This example shows a few examples of how this can be done.

```
*
* Combining Edit Symbols.
*
 identification division.
 program-id. EXAMPLE-32.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01   a     pic s9999v99 value is -123.40.
 01   b     pic s999999v99 value is 12345.60.

 01   c     pic $z(6).99.
 01   d     pic $zz,zzz,zzz.99CR.
 01   e     pic $zz,zzz,zz9.99CR.
 01   f     pic $zz,zzz,zz9.99CR.

 procedure division.
      add a b giving c.
      subtract a from b giving d.
      multiply a by b giving e.
      divide a into b giving f.
      display 'a ' a.
      display 'b ' b.
      display ' '.
      display 'c ' c.
      display 'd ' d.
      display 'e ' e.
      display 'f ' f.
      stop run.
```

Sample Program Execution

```
run
Execution Begins...
a  01234}
b  0123456{

c  $ 12222.20
d  $       12,469.00
e  $  1,523,447.04CR
f  $           100.04CR
...Execution ends.
```

Notes

(1) The '$' and 'z' symbol can be combined in one **picture** clause as in the above cases. In these cases the '$' signs all occur in column one of the output. In previous example the '$' signs *floated* to be immediately left of the most significant digit.

(2) In the last two cases a '9' has been inserted to the left of the decimal. This would cause a value, say .25, to be displayed as 0.25 instead of .25.

(3) The use of various edit characters to produce different forms of output depends very much on the user's particular preferences. It may also depend on installation standards.

1.7 Two Examples Using Files and Arithmetic.

1.7.1 Student Averages

This and the next example use some of the material presented in the previous examples to demonstrate how arithmetic can be used.

```
*
* Calculate Student Averages.
*
 identification division.
 program-id. EXAMPLE-33.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
        select student-file
              assign to 'textfile'.
        select screen
              assign to 'terminal'.

 data division.

 file section.
 fd     student-file
        label records are standard.
 01     student-record.
        02 filler          pic x(60).

 fd     screen
        label records are standard.
 01     display-record.
        02 filler          pic x(80).

 working-storage section.

 01     student-total                pic 9(5).
```

```
01  student-data.
    02 student-no          pic xxxx.
    02 name                pic x(20).
    02 age                 pic xx.
    02 sex                 pic x.
    02 class               pic x.
    02 school              pic x.
    02 algebra             pic 999.
    02 geometry            pic 999.
    02 physics             pic 999.
    02 chemistry           pic 999.
    02 english             pic 999.

01  report-heading.
    02 filler          pic x(20) value is spaces.
    02 filler          pic x(30) value is 'Student Averages'.
    02 filler          pic x(30) value is spaces.

01  first-line.
    02 out-student-no      pic x(4).
    02 filler              pic x(5) value is spaces.
    02 out-name            pic x(20).
    02 out-algebra         pic z(4)9.
    02 out-geometry        pic z(4)9.
    02 out-physics         pic z(4)9.
    02 out-chemistry       pic z(4)9.
    02 out-english         pic z(4)9.
    02 out-average         pic z(5).9.

01  blank-line             pic x(80) value is spaces.

procedure division.
    open input student-file.
    open output screen.
    write display-record from report-heading.
    write display-record from blank-line.
    perform read-student-record.
    perform process-student-data
        until student-no = high-values.
    close student-file
          screen.
    stop run.
```

```
process-student-data.
    perform process-student-marks.
    perform display-student-data.
    perform read-student-record.

process-student-marks.
    move zero to student-total.
    add algebra to student-total.
    add geometry to student-total.
    add physics to student-total.
    add chemistry to student-total.
    add english to student-total.
    divide student-total by 5 giving out-average.

display-student-data.
    move student-no to out-student-no.
    move name to out-name.
    move algebra to out-algebra.
    move geometry to out-geometry.
    move physics to out-physics.
    move chemistry to out-chemistry.
    move english to out-english.
    write display-record from first-line.
    write display-record from blank-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                                Student Averages

1234    Smith          SA    75  100   75   65   84   79.8

1236    Jones          TO    76   78   55   57   78   68.8

1238    Winterbourne   MS    78   88   56   67   88   75.4

1239    Harrison       K     22   87   65   87   68   65.8

1240    Graham         JW     0   68   75   67   87   59.4

1242    Welch          JW    75   75   76   75   75   75.2

1243    Dirksen        PH    74   85   54   68   84   73.0

1245    Cowan          DD    55   66   77   88   99   77.0

1249    Sullivan       J     44   55   66   77   88   66.0

1256    Kitchen        MP    74   49  100   97   36   71.2

1266    Taylor         YO    95   83   72   66   55   74.2

1268    Allen          TT    98   84   73   65   59   75.8

1270    Xerxes         X     99   88   77   66   55   77.0

1272    Zimmerman      AB    95   85   78   61   57   75.2

1375    Quantas        FL    66   66   66   66   66   66.0

1388    Beatle         RA    65   62   73   76   87   72.6

1390    Cruikshank     TR    55   64   77   76   85   71.4

1393    Hopper         BU    45   69   37   26   35   42.4

...Execution ends.
```

Notes

(1) This example calculates the average mark of the five courses for each student and displays a report of this information.

(2) A heading is displayed for the report and then one line is displayed for each student containing the marks and the calculated average.

(3) An area called "student-total" is defined in working storage to contain the total of the five marks.

(4) The definitions for the five marks in "student-data" have been changed from **pic** xxx to **pic** 999. These values will be used in arithmetic operations and hence must be defined with 9's instead x's.

(5) The calculation of the student average is done in the "process-student-marks" paragraph using a number of sentences each with one verb. A slightly different version of the **add** is used, namely

```
add algebra to student-total
```

instead of the longer and more complicated

```
add algebra student-total giving student-total.
```

(6) The required computation could have been done in the following manner

```
compute out-average = (algebra + geometry + physics
                        + chemistry + english) / 5.
```

or alternatively as

```
add algebra geometry physics chemistry english
            giving student-total.
divide student-total by 5 giving out-average.
```

(7) If for some reason any particular mark in the file contained non-numeric characters, an error would occur when the value was read and the program would terminate.

1.7.2 School Algebra Averages.

This example calculates school algebra averages.

```
*
* Calculate Class Algebra Averages.
*
 identification division.
 program-id. EXAMPLE-34.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
         assign to 'textfile'.
     select screen
         assign to 'terminal'.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler         pic x(60).

 fd  screen
     label records are standard.
 01  display-record.
     02 filler         pic x(80).

 working-storage section.

 01  totals.
     02 total-1                 pic 9(5).
     02 total-2                 pic 9(5).
     02 total-3                 pic 9(5).
     02 count-1                 pic 9(5).
     02 count-2                 pic 9(5).
     02 count-3                 pic 9(5).
```

```
01   student-data.
     02 student-no            pic xxxx.
     02 name                  pic x(20).
     02 age                   pic xx.
     02 sex                   pic x.
     02 class                 pic x.
     02 school                pic x.
     02 algebra               pic 999.
     02 geometry              pic 999.
     02 physics               pic 999.
     02 chemistry             pic 999.
     02 english               pic 999.

01   report-heading.
     02 filler            pic x(20) value is spaces.
     02 filler            pic x(30) value is 'Algebra Averages'.
     02 filler            pic x(30) value is spaces.

01   school-line.
     02 filler                pic x(8) value is 'School'.
     02 out-school            pic 9.
     02 filler                pic x(12) value is ' Average is '.
     02 out-average           pic z(5).9.

01   blank-line               pic x(80) value is spaces.

procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     move zeros to totals.
     perform process-student-data
          until student-no = high-values.
     perform display-averages
     close student-file
             screen.
     stop run.

process-student-data.
     perform process-student-marks.
     perform read-student-record.
```

```
process-student-marks.
    if school = '1'
        add algebra to total-1
        add 1 to count-1
    else if school = '2'
        add algebra to total-2
        add 1 to count-2
    else
        add algebra to total-3
        add 1 to count-3.

display-averages.
    move 1 to out-school.
    divide total-1 by count-1 giving out-average.
    write display-record from school-line.
    move 2 to out-school.
    divide total-2 by count-2 giving out-average.
    write display-record from school-line.
    move 3 to out-school.
    divide total-3 by count-3 giving out-average.
    write display-record from school-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                     Algebra Averages

School  1 Average is  63.2
School  2 Average is  62.8
School  3 Average is  71.1
...Execution ends.
```

Notes

(1) This example calculates the algebra average for each of the three schools.

(2) In this case, three areas are defined to contain the total of the algebra marks for each school. Three other areas are defined to contain the number of students in each school.

(3) The six totals have been defined as a data structure called "totals". This permits us to store zeros in each of the six areas with the one sentence

```
move zeros to totals
```

instead of requiring six separate **move** sentences.

(4) The "process-student-marks" paragraph determines the school of the current record and then adds the algebra mark to the appropriate total. It also adds 1 to the appropriate "count" total.

(5) After all the records are read, the averages are calculated and displayed in the "display-averages" paragraph.

(6) The reader might be tempted to use **value is** clauses to set the six totals to zero. This would work correctly in this example. However, it not considered good programming practice to initialize values in working storage unless they are to remain unchanged by the program. In this case the totals were changed and hence were zeroed by the appropriate **move.**

1.8 Subscripted Data-names.

1.8.1 Subscripted Data-names.

Most data processing applications in some way involve the use of *tables* of information. This section introduces features of COBOL which permit the easy handling of table data, and in particular *subscripted data-names* are introduced.

```
*
* Number of Students at each School.
*
 identification division.
 program-id. EXAMPLE-35.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
          assign to 'textfile'.
     select screen
          assign to 'terminal'.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler        pic x(60).

 fd  screen
     label records are standard.
 01  display-record.
     02 filler        pic x(80).

 working-storage section.

 01  i        pic 9.
```

```
01  count-table.
    02 counts      pic 9(5) occurs 3 times.

01  student-data.
    02  student-no           pic xxxx.
    02 name                  pic x(20).
    02  age                  pic xx.
    02  sex                  pic x.
    02 class                 pic x.
    02 school                pic x.
    02  algebra              pic 999.
    02  geometry             pic 999.
    02 physics               pic 999.
    02 chemistry             pic 999.
    02 english               pic 999.

01  report-heading.
    02 filler            pic x(20) value is spaces.
    02 filler            pic x(30) value is 'Enrollment Numbers'.
    02 filler            pic x(30) value is spaces.

01  school-line.
    02 filler                pic x(8) value is 'School '.
    02  out-school           pic 9.
    02 filler                pic x(5) value is ' has '.
    02 out-count             pic zz9.
    02 filler                pic x(9) value is ' Students'.

01  blank-line               pic x(80) value is spaces.
```

```
procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     move zeros to count-table.
     perform process-student-data
          until student-no = high-values.
     move 1 to i.
     perform display-totals 3 times.
     close student-file
            screen.
     stop run.

process-student-data.
     perform process-student-marks.
     perform read-student-record.

process-student-marks.
     move school to i.
     add 1 to counts(i).

display-totals.
     move i to out-school.
     move counts(i) to out-count.
     write display-record from school-line.
     add 1 to i.

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                         Enrollment Numbers

School  1 has   5 Students
School  2 has   6 Students
School  3 has   7 Students
...Execution ends.
```

Notes

(1) This program reads the student file and determines the number of students in each of the three schools. Note that a previous example needed this information in order to calculate school algebra averages. In that case, three areas were defined to hold the three required counters. In this example we introduce a new way of defining the three counters.

(2) A new definition appears in working storage, namely,

```
01  count-table.
    02 counts pic 9(5) occurs 3 times.
```

We have introduced the **occurs** clause on the 02-level table entry item. This means that the data-name "counts" is defined 3 times, each time with the same **picture** clause of 9(5). The 3 entries in the table are referred to as follows:

```
counts(1)
counts(2)
counts(3)
```

The integers contained in parentheses are known as *subscripts*.

(3) The data-name "counts" must be associated with one of the valid subscripts to be meaningful. The value of the subscript must be in the range 1-3 i.e. by using the clause

```
occurs 3 times
```

we promise that we will not use a subscript greater than 3. COBOL does not permit a subscript of zero or any negative value.

(4) The data-name "count-table" can be used to refer to all the items in the table. Thus the statement

```
move zeros to count-table
```

sets each item in the table to zero.

(5) The paragraph

```
process-student-marks.
     move school to i.
     add 1 to counts(i).
```

causes 1 to be added to the appropriate table item. The data-name "i" is defined as a numeric item and the value of "school" is moved to "i". Recall that the school field of the student record contains either a 1, 2, or 3. Subscripts must be defined as numeric fields and must contain numeric values.

(6) We could have defined "school" with a **picture** of '9' instead of 'x' and then the paragraph could have been written as

```
process-student-marks.
     add 1 to counts(school).
```

(7) In order to display the calculated values, "i" is initialized to 1 and the "display-totals" paragraph is executed 3 times; each time the appropriate line is displayed and "i" is incremented by 1.

(8) The power of the use of subscripts should be self-evident. For example if the number of schools were increased from 3 to 9, the "process-student-marks" paragraph requires no modification. Of course, the **occurs** clause would have to be changed to define nine items. We would also have to modify the part of the program that displays the results.

(9) The user may have been tempted to use the **value is** clause to set the initial table values to zero. This is not possible as COBOL does *not* permit one to use the **value is** clause to initialize subscripted data-names.

1.8.2 Perform Varying.

When using subscripted data-names, we quite often wish to execute a paragraph repetitively; the only difference is that we wish to change the subscript's value. This was the case in the previous example when we were displaying the results. The **perform** verb with the **varying** option gives us this capability.

```
*
* School Algebra Averages using Subscripted Data-names
* and the Perform Varying.
*
 identification division.
 program-id. EXAMPLE-36.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).

 working-storage section.

 01   i          pic 9.
```

```
    01  totals-table.
        02 total        pic 9(5) occurs 3 times.
        02 counts       pic 9(5) occurs 3 times.

    01  school-table.
        02 school-data       pic x(10) occurs 3 times.

    01  student-data.
        02 student-no            pic xxxx.
        02 name                  pic x(20).
        02 age                   pic xx.
        02 sex                   pic x.
        02 class                 pic x.
        02 school                pic x.
        02 algebra               pic 999.
        02 geometry              pic 999.
        02 physics               pic 999.
        02 chemistry             pic 999.
        02 english               pic 999.

    01  report-heading.
        02 filler           pic x(20) value is spaces.
        02 filler           pic x(30) value is 'Algebra Averages'.
        02 filler           pic x(30) value is spaces.

    01  school-line.
        02 filler                pic x(13) value is ' Average for '.
        02 out-school            pic x(10).
        02 filler                pic x(4) value is ' is '.
        02 out-average           pic z(5).9.

    01  blank-line               pic x(80) value is spaces.
```

```
procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     move 'Central' to school-data(1).
     move 'Western' to school-data(2).
     move 'Southern' to school-data(3).
     perform read-student-record.
     move zeros to totals-table.
     perform process-student-data
          until student-no = high-values.
     perform display-averages
          varying i from 1 by 1
          until i > 3.
     close student-file
            screen.
     stop run.

process-student-data.
     perform process-student-marks.
     perform read-student-record.

process-student-marks.
     move school to i.
     add algebra to total(i).
     add 1 to counts(i).

display-averages.
     move school-data(i) to out-school.
     divide total(i) by counts(i) giving out-average.
     write display-record from school-line.

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                                     Algebra Averages

  Average for Central            is   63.2
  Average for Western            is   62.8
  Average for Southern           is   71.1
...Execution ends.
```

Notes

(1) This example calculates average algebra mark for each of the three schools. It also displays a school name instead of a school number.

(2) Another table is introduced to hold the sum of the grades for each school. While it could have been defined as a separate table, it has been defined as part of the totals-table. This permits us to zero both tables with one **move** sentence.

(3) The program reads each record and accumulates the algebra mark for the appropriate school as well as incrementing the appropriate counter.

(4) The "display-averages" paragraph calculates the average for each school and displays the desired values.

(5) The sentence

```
perform display-averages
     varying i from 1 by 1
     until i > 3.
```

causes the "display-averages" paragraph to be executed repeatedly as long as i is not greater than 3. Before the paragraph is executed, the value of "i" is set to 1 and the condition is evaluated to determine if it is true or false. If it is true the paragraph is executed; if false control passes to the next sentence. Each successive time, "i" is incremented by 1 and the condition is tested to determine if the paragraph should be executed.

(6) We could have displayed the results in the reverse order by changing the **perform** as follows:

```
perform display-averages
     varying i from 3 by -1
     until i = 0.
```

In this case, "i" will start with the value 3, and each time will be decremented by 1 until "i" is zero.

(7) A complete description of the **perform** verb can be found in the reference manual.

(8) Another table has been defined to contain the names of the three schools. This table is initialized with three **moves** of 'Central', 'Western', and 'Southern' respectively. These names are **moved** to the display line in the "display-averages" paragraph. The next example will show how to define these initial values in working storage.

(9) Note that tables can be defined using x's as well as 9's in the **picture** clauses.

1.8.3 The Redefines Clause with Subscripted Data-names.

While COBOL does not permit the initialization of tables directly using the **value is** clause, an indirect method is available.

```
*
* Redefines Clause.
*
 identification division.
 program-id. EXAMPLE-37.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
         assign to 'textfile'.
     select screen
         assign to 'terminal'.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler        pic x(60).

 fd  screen
     label records are standard.
 01  display-record.
     02 filler        pic x(80).

 working-storage section.

 01  i       pic 9.

 01  totals-table.
     02 total          pic 9(5) occurs 3 times.
     02 counts        pic 9(5) occurs 3 times.
```

```
01   school-table-data.
     02 filler          pic x(10) value is 'Central'.
     02 filler          pic x(10) value is 'Western'.
     02 filler          pic x(10) value is 'Southern'.

01   school-table redefines school-table-data.
     02 school-data           pic x(10) occurs 3 times.

01   student-data.
     02  student-no               pic xxxx.
     02 name                      pic x(20).
     02  age                      pic xx.
     02  sex                      pic x.
     02  class                    pic x.
     02 school                    pic x.
     02  algebra                  pic 999.
     02  geometry                 pic 999.
     02 physics                   pic 999.
     02 chemistry                 pic 999.
     02  english                  pic 999.

01   report-heading.
     02 filler               pic x(20) value is spaces.
     02 filler               pic x(30) value is 'Algebra Averages'.
     02 filler               pic x(30) value is spaces.

01   school-line.
     02 filler                    pic x(13) value is ' Average for '.
     02  out-school               pic x(10).
     02 filler                    pic x(4) value is ' is '.
     02  out-average              pic z(5).9.

01   blank-line                   pic x(80) value is spaces.
```

```
procedure division.
    open input student-file.
    open output screen.
    write display-record from report-heading.
    write display-record from blank-line.
    perform read-student-record.
    move zeros to totals-table.
    perform process-student-data
        until student-no = high-values.
    perform display-averages
        varying i from 1 by 1
        until i > 3.
    close student-file
          screen.
    stop run.

process-student-data.
    perform process-student-marks.
    perform read-student-record.

process-student-marks.
    move school to i.
    add algebra to total(i).
    add 1 to counts(i).

display-averages.
    move school-data(i) to out-school.
    divide total(i) by counts(i) giving out-average.
    write display-record from school-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                       Algebra Averages

 Average for Central        is   63.2
 Average for Western        is   62.8
 Average for Southern       is   71.1
...Execution ends.
```

Notes

(1) This example also calculates and displays school algebra averages.

(2) An area called "school-table-data" is initialized to contain the three names of the schools.

(3) Immediately following the definition of the "school-table-data", we have inserted the following description of "school-table".

```
01 school-table redefines school-table-data.
   02 school-data pic x(10) occurs 3 times.
```

This differs from the previous example in that the **redefines** clause is introduced. This means that the item "school-table" occupies the *same* area in working storage as the item "school-table-data".

(4) Both areas are defined to have 30 characters; COBOL does not require that they same length if both are 01-level items.

(5) In essence, the **redefines** permits the user to give two or more definitions to the same area in working storage and then to refer to the area with the appropriate data-name.

(6) There are many other applications of this feature, but the assignment of initial values in a table of subscripted data-names is an important one.

1.8.4 Tables with Two Subscripts.

Consider the problem of finding the number of students in each of the four classes at each of the three schools. In this case it is more convenient to set up a table containing three rows representing the schools and four columns representing the classes. This example introduces tables with two subscripts.

```
*
* School Course Averages
* Data-names with Two Subscripts.
*
 identification division.
 program-id. EXAMPLE-38.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
        select student-file
             assign to 'textfile'.
        select screen
             assign to 'terminal'.

 data division.

 file section.
 fd     student-file
        label records are standard.
 01     student-record.
        02 filler          pic x(60).

 fd     screen
        label records are standard.
 01     display-record.
        02 filler          pic x(80).

 working-storage section.

 01     i          pic 9.
 01     j          pic 9.
```

```
01   school-table-data.
     02 filler          pic x(10) value is 'Central'.
     02 filler          pic x(10) value is 'Western'.
     02 filler          pic x(10) value is 'Southern'.

01   school-table redefines school-table-data.
     02 school-data          pic x(10) occurs 3 times.

01   summary-table.
     02 count-by-school occurs 3 times.
        03count-by-class             pic 9(5) occurs 4 times.

01   student-data.
     02  student-no          pic xxxx.
     02 name                 pic x(20).
     02  age                 pic xx.
     02  sex                 pic x.
     02  class               pic x.
     02 school               pic x.
     02  algebra             pic 999.
     02  geometry            pic 999.
     02 physics              pic 999.
     02 chemistry            pic 999.
     02  english             pic 999.

01   report-heading.
     02 filler             pic x(20) value is spaces.
     02 filler             pic x(30) value is 'Enrollment Table'.
     02 filler             pic x(30) value is spaces.

01   school-line.
     02  out-school          pic x(10).
     02  out-counts          pic z(4)9 occurs 4 times.

01   blank-line              pic x(80) value is spaces.
```

```
procedure division.
    open input student-file.
    open output screen.
    write display-record from report-heading.
    write display-record from blank-line.
    perform read-student-record.
    move zeros to summary-table.
    perform process-student-data
        until student-no = high-values.
    perform display-averages
        varying i from 1 by 1
        until i > 3.
    close student-file
          screen.
    stop run.

process-student-data.
    perform process-student-marks.
    perform read-student-record.

process-student-marks.
    move school to i.
    move class to j.
    add 1 to count-by-class(i, j).

display-averages.
    move school-data(i) to out-school.
    perform move-counts
        varying j from 1 by 1
        until j > 4.
    write display-record from school-line.

move-counts.
    move count-by-class(i, j) to out-counts(j).

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                              Enrollment Table

Central              1   2   2   0
Western              0   2   1   3
Southern             2   1   3   1
...Execution ends.
```

Notes

(1) A more involved definition is required to set up a table with two subscripts, namely

```
01  summary-table.
      02 count-by-school occurs 3 times.
          03 count-by-class pic 9(5) occurs 4 times.
```

(2) Note that "count-by-school" is defined to occur 3 times because there are 3 schools. These three entries are referred to as:

```
count-by school(1)
count-by-school(2)
count-by-school(3)
```

(3) For each class, we have "count-by-class" which occurs 4 times, once for each class. A problem arises when we wish to reference the data-names "count-by-class". If we consider the example

```
count-by-class(3)
```

we have a vague reference; we know it means the third entry, but for which class? The problem is solved by introducing a *second subscript as follows*

```
count-by-class(3, 2)
```

The first subscript refers to school 3, and the second one refers to class 2.

(4) Every reference to the data-name "count-by-class" *must* have two subscripts and is referred to as a *doubly-subscripted* data-name. "Summary-table" is often referred to as a *two-dimensional table* or an *array*.

(5) The above definition can be thought to have created 12 items each with a **picture** 9(5). The twelve items can be zeroed with the sentence

```
move zeros to summary-table.
```

(6) The paragraph "process-student-marks" assigns the school to "i" and the class to "j". Then "i" and "j" are used as row and column subscripts respectively and one is added to the correct table entry.

(7) After all the records are processed, the table is displayed. The area "school-line" was modified to contain a table to hold the four class counts. The values from a row are obtained from the "summary-table and then the record is displayed. This action is repeated for each row.

(8) COBOL allows up to three subscripts to be used.

1.9 Relative Files

1.9.1 Create a Relative File.

Suppose that a teacher wished to display the record for a particular student from the student file in order to look at the student's marks. It would not be difficult to write a program to accomplish this. However, consider now that the teacher wished to examine a second student's record. If this record occurred after the one just examined, there would be no major problem to retrieve it and display it. However, if it occurred before, we would have to go to some extra effort to retrieve the new record. The difficulty lies in the fact that the student file is a sequential file. *Relative files* permit us to access any record as easily as any other record. This section introduces relative files.

```
*
* Create a Relative File.
*
 identification division.
 program-id. EXAMPLE-39.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
         assign to 'textfile'.
     select student-inquiry
         assign to 'dirfile,rel'
         organization is relative
         access is sequential.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler         pic x(60).
```

```
fd    student-inquiry
      label records are standard.
01    direct-record.
      02 filler          pic x(60).

working-storage section.

01    student-data.
      02  student-no            pic xxxx.
      02 name                   pic x(20).
      02  age                   pic xx.
      02  sex                   pic x.
      02  class                 pic x.
      02 school                 pic x.
      02  algebra               pic 999.
      02  geometry              pic 999.
      02  physics               pic 999.
      02  chemistry             pic 999.
      02  english               pic 999.

procedure division.
      open input student-file
           output student-inquiry.
      perform read-student-record.
      perform process-student-data
          until student-no = high-values.
      close student-file
            student-inquiry.
      stop run.

process-student-data.
      write direct-record from student-data.
      perform read-student-record.

read-student-record.
      read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
...Execution ends.
```

Notes

(1) This program reads the student file and creates a new version of the file as a *relative file.*

(2) A new file called "student-inquiry" is defined as a relative file. Its system-name is "dirfile,rel". The *rel* is required by the CBM-SuperPET to indicate that the file is relative. The chapter containing system dependent information should be consulted regarding the format of file-names on other systems.

(3) Two extra clauses

```
organization is relative
access is sequential
```

are added to the **select** statement. The first indicates that the file will be used in the future as a relative file. The second indicates that for this program we will be writing the records in a sequential fashion. This may seem somewhat confusing but we wish to read the records from the sequential file and write them in the same order, sequentially, as a new relative file. The next example will use this new file as a relative file.

(4) The **fd** for the new file is the same as the **fd** for the student file, except it has a new file and record name.

(5) Opening and closing the relative file is done in the same manner as opening and closing sequential files.

(6) Writing records to a relative file is also done in the same manner as writing records to a sequential file.

1.9.2 Read a Relative File.

Having written a relative file, we now wish to assure ourselves that the file exists and further that we can use it in a 'relative' fashion.

```
*
* Read a Relative File.
*
 identification division.
 program-id. EXAMPLE-40.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-inquiry
         assign to 'dirfile,rel'
         organization is relative
         access is random
         relative key is student-key.

 data division.

 file section.

 fd  student-inquiry
     label records are standard.
 01  direct-record.
     02 filler       pic x(60).

 working-storage section.

 01  student-key      pic 999.
```

```
01  student-data.
    02 student-no          pic xxxx.
    02 name                pic x(20).
    02 age                 pic xx.
    02 sex                 pic x.
    02 class               pic x.
    02 school              pic x.
    02 algebra             pic 999.
    02 geometry            pic 999.
    02 physics             pic 999.
    02 chemistry           pic 999.
    02 english             pic 999.

procedure division.
    open input student-inquiry.
    move 18 to student-key.
    perform read-student-record.
    perform process-student-data
        until student-key = 999.
    close student-inquiry.
    stop run.

process-student-data.
    display student-data.
    subtract 1 from student-key.
    perform read-student-record.

read-student-record.
    read student-inquiry into student-data
        invalid key move 999 to student-key.
```

Sample Program Execution

```
run
Execution begins...
1393Hopper          BU 15f23045069037026035
1390Cruikshank      TR 15f33055064077076085
1388Beatle          RA 15f11065062073076087
1375Quantas         FL 15m22066066066066066
1272Zimmerman       AB 13f32095085078061057
1270Xerxes          X  13f13099088077066055
1268Allen           TT 13f21098084073065059
1266Taylor          YO 13f33095083072066055
1256Kitchen         MP 14m43074049100097036
1249Sullivan        J  15f42044055066077088
1245Cowan           DD 15f33055066077088099
1243Dirksen         PH 14m42074085054068084
1242Welch           JW 14m31075075076075075
1240Graham          JW 14m21000068075067087
1239Harrison        K  14m42022087065087068
1238Winterbourne    MS 14m31078088056067088
1236Jones           TO 14m22076078055057078
1234Smith           SA 14m13075100075065084
...Execution ends.
```

Notes

(1) This example reads and displays the 18 records of the relative file in reverse order i.e. the last one first, and the first one last.

(2) The **select** clause is modified to indicate that

access is random

instead of sequential i.e. we can now access the records in any order.

(3) Another clause

```
relative key is student-key
```

is added to the **select** clause. "Student-key" is a data-name which will be used to indicate which record we wish to read. Note that is defined in *working storage and must be defined using a* ***picture*** *with 9's. The value* of "student-key" must be an integer value lying in the range 1 to the 'number' of records in the relative file, in our case 18.

(4) This value is used to read the desired record in the file. If the value of "student-key" were 7, then the seventh record would be read.

(5) The **read** sentence has also been changed to reflect that we are reading a relative file.

```
read student-inquiry into student-data
     invalid key move 999 to student-key.
```

The **at end** clause is not used when reading a relative file. The clause

```
invalid key move 999 to student-key
```

is executed only if an invalid key is used, in our case outside the 1-18 range.

(6) When the program is executed, the value of 18 is assigned to "student-key" which will result in the last record to being read. Then "student-key" is decremented by one and the second-last record is read. This process continues until "student-key" is one and the first record is read. "Student-key is again decremented and now has a value of zero. When we attempt to read the zero'th record the **invalid key** clause is executed and 999 is assigned to "student-key" which in turn causes the "process-student-data" paragraph to terminate.

(7) Earlier we suggested that one should use **high-values** instead of a constant such as 999 to signal that there were no more records or in this case that an **invalid key** was encountered. **High-values** can only be assigned to a field whose **picture** clause contains x's; "student-key" must be defined using 9's. Hence, we use 999 to signal that the use of an invalid key. It is suggested that the "student-key" field be made at least one position larger than the number of records in the file.

1.9.3 Create a Relative File with an Index.

In order to use a relative file, we have to know the position of the record in the file. It would seem more appropriate to use the student number to identify records. In order to do this we must create an *index* to the file.

```
*
* Read a Relative File.
*
 identification division.
 program-id. EXAMPLE-41.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-inquiry
         assign to 'dirfile,rel'
         organization is relative
         access is random
         relative key is student-key.
     select index-file
         assign to 'indfile'.

 data division.

 file section.

 fd  student-inquiry
     label records are standard.
 01  direct-record.
     02 filler          pic x(60).

 fd  index-file
     label records are standard.
 01  index-record.
     02 filler          pic xxxx.

 working-storage section.

 01  student-key          pic 999.
```

```
01   student-data.
     02  student-no          pic xxxx.
     02 name                 pic x(20).
     02  age                 pic xx.
     02  sex                 pic x.
     02 class                pic x.
     02 school               pic x.
     02  algebra             pic 999.
     02  geometry            pic 999.
     02 physics              pic 999.
     02 chemistry            pic 999.
     02 english              pic 999.

procedure division.
     open input student-inquiry
          output index-file.
     move 1 to student-key.
     perform read-student-record.
     perform create-student-index
          until student-key = 999.
     close student-inquiry
            index-file.
     stop run.

create-student-index.
     write index-record from student-no.
     add 1 to student-key.
     perform read-student-record.

read-student-record.
     read student-inquiry into student-data
          invalid key move 999 to student-key.
```

Sample Program Execution

```
run
Execution begins...
...Execution ends.
```

Notes

(1) This example reads the "student-inquiry" file and creates a new file called "index-file". This new file will also contain 18 records with each record containing only the student number.

(2) The program causes each record starting with the first to be read, and a new record containing the student number is written.

1.9.4 Extract Records from a Relative File.

Having created an *index*, we now wish to use it retrieve records randomly from the student file using the student number as the key.

```
*
* Read the Index and Selective Records.
*
 identification division.
 program-id. EXAMPLE-42.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select index-file
         assign to 'indfile'.
     select student-inquiry
         assign to 'dirfile,rel'
         organization is relative
         access is random
         relative key is student-key.

 data division.

 file section.

 fd  student-inquiry
     label records are standard.
 01  direct-record.
     02 filler           pic x(60).

 fd  index-file
     label records are standard.
 01  index-record.
     02 filler           pic xxxx.
```

```
working-storage section.

01  student-key          pic 999.
01  student-found        pic xxx.
01  student-id           pic xxxx.
01  index-exists         pic xxx.
01  number-of-keys       pic 99.
01  i                    pic 999.

01  student-index.
    02 index-key      pic xxxx occurs 18 times.

01  student-data.
    02 student-no        pic xxxx.
    02 name              pic x(20).
    02 age               pic xx.
    02 sex               pic x.
    02 class             pic x.
    02 school            pic x.
    02 algebra           pic 999.
    02 geometry          pic 999.
    02 physics           pic 999.
    02 chemistry         pic 999.
    02 english           pic 999.

procedure division.
    perform read-index.
    if index-exists = 'yes'
        perform display-student-records.
    stop run.

display-student-records.
    open input student-inquiry.
    perform get-student-id.
    perform process-student-records
        until student-id = 'stop'.
    close student-inquiry.
```

```
process-student-records.
    perform find-student-key.
    if student-found = 'yes'
        read student-inquiry into student-data
        display student-data
    else
        display 'invalid student id. ' student-id.
    perform get-student-id.

get-student-id.
    display 'enter student id - stop to stop'.
    accept student-id.

find-student-key.
    move 'no' to student-found.
    perform find-key
         varying i from 1 by 1
         until i > number-of-keys or student-found = 'yes'.

find-key.
    if student-id =index-key(i)
        move i to student-key
        move 'yes' to student-found.

read-index.
    open input index-file.
    move 'no' to index-exists.
    move 0 to i.
    perform read-index-record.
    perform store-and-read-index
        until student-id = high-values.
    if i > 0
        move i to number-of-keys
        move 'yes' to index-exists.
    close index-file.

store-and-read-index.
    add 1 to i.
    move student-id to index-key(i)
    perform read-index-record.
```

```
read-index-record.
    read index-file into student-id
        at end move high-values to student-id.
```

Sample Program Execution

```
run
Execution begins...
enter student id - stop to stop
1234
1234Smith          SA 14m13075100075065084
enter student id - stop to stop
1393
1393Hopper         BU 15f23045069037026035
enter student id - stop to stop
5427
invalid student id. 5427
enter student id - stop to stop
stop
...Execution ends.
```

Notes

(1) This example asks the user to enter a student number. Then the record for that student is displayed. The program terminates when a value of 'stop' is entered. If a student number which is not in the file is entered, a message indicating this is displayed and the program requests another student number to be entered.

(2) The program initially reads the "index-file" and creates a table with one entry for each record in the file. If no records exist in the "index-file", the program terminates.

(3) The user is then requested to enter a student number and this number is successively compared to each value in the table. When an equal comparison is found, the position of the item in the table is used as the position of the record in the relative file. If the number is not found in the table, a signal is set and a message will be displayed.

(4) The method of searching the table is called a *sequential search*. If the table were quite large, it would probably be better to use some other searching method or algorithm. These methods are described in many computer science textbooks.

(5) If the number of records were quite large, there might not be enough memory to hold the entire index table. In this case, one would have to set up a scheme to possibly have an index to the index.

(6) It would be possible in this example to re-write the student record if, for example, we wished to change a mark or even add a field with a new mark. Thus we could read the record, make any changes, and then re-write the record in the same place using the same value of "student-key"; the value is not changed by the read. In this case, the file should be opened using the **I-O** option instead of the **input** option.

(7) It is also possible to add new records to the file. However, the method to do this is 'system dependent'. The user is referred to the reference manual to determine how this can be done.

1.10 Miscellaneous.

1.10.1 Create the Student File.

This program creates the student file used in many of the examples in the tutorial section of this text.

```
*
* Create Demonstration File.
*
 identification division.
 program-id. EXAMPLE-43.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
     select student-file
         assign to 'textfile'.

 data division.

 file section.
 fd  student-file
     label records are standard.
 01  student-record.
     02 filler        pic x(60).

 working-storage section.

 01  rec-number           pic 999.

 01  student-data.
     02  filler        pic x(60).
```

```
01  demo-file.
    02  demo-data.
        03 filler  pic x(60) value is
             '1234Smith                SA 14m13075100075065084'.
        03 filler  pic x(60) value is
             '1236Jones                TO 14m22076078055057078'.
        03 filler  pic x(60) value is
             '1238Winterbourne         MS 14m31078088056067088'.
        03 filler  pic x(60) value is
             '1239Harrison             K  14m42022087065087068'.
        03 filler  pic x(60) value is
             '1240Graham               JW 14m21000068075067087'.
        03 filler  pic x(60) value is
             '1242Welch                JW 14m31075075076075075'.
        03 filler  pic x(60) value is
             '1243Dirksen              PH 14m42074085054068084'.
        03 filler  pic x(60) value is
             '1245Cowan                DD 15f33055066077088099'.
        03 filler  pic x(60) value is
             '1249Sullivan             J  15f42044055066077088'.
        03 filler  pic x(60) value is
             '1256Kitchen              MP 14m43074049100097036'.
        03 filler  pic x(60) value is
             '1266Taylor               YO 13f33095083072066055'.
        03 filler  pic x(60) value is
             '1268Allen                TT 13f21098084073065059'.
        03 filler  pic x(60) value is
             '1270Xerxes               X  13f13099088077066055'.
        03 filler  pic x(60) value is
             '1272Zimmerman            AB 13f32095085078061057'.
        03 filler  pic x(60) value is
             '1375Quantas              FL 15m22066066066066066'.
        03 filler  pic x(60) value is
             '1388Beatle               RA 15f11065062073076087'.
        03 filler  pic x(60) value is
             '1390Cruikshank           TR 15f33055064077076085'.
        03 filler  pic x(60) value is
             '1393Hopper               BU 15f23045069037026035'.
    02  demo-info redefines demo-data.
        03demo-rec      pic x(60) occurs 18 times.
```

```
procedure division.
    open output student-file.
    perform write-demo
        varying rec-number from 1 by 1
        until rec-number > 18.
    close student-file.
    stop run.

write-demo.
    move demo-rec(rec-number) to student-data.
    write student-record from student-data.
```

Sample Program Execution

```
run
Execution begins...
...Execution ends.
```

Waterloo microCOBOL

Reference Manual

Waterloo Computing Systems Newsletter

The software described in this manual was implemented by Waterloo Computing Systems Limited. From time-to-time enhancements to this system or completely new systems will become available.

A newsletter is published periodically to inform users of recent developments in Waterloo software. This publication is the most direct means of communicating up-to-date information to the various users. Details regarding subscriptions to this newsletter may be obtained by writing:

Waterloo Computing Systems Newsletter
Box 943,
Waterloo, Ontario, Canada
N2J 4C3

Chapter 2

Structure of a COBOL Program

2.1 Overview

COBOL (**CO**mmon **B**usiness **O**riented **L**anguage) is a computer programming language specifically designed for use in solving business problems. Waterloo microCOBOL is intended to be an implementation of part of the accepted standard for COBOL (ANSI X3.23-1974). For persons familiar with this COBOL standard, the language intended to be supported includes level one of the **NUCLEUS, SEQUENTIAL I-O, RELATIVE I-O** and **TABLE-HANDLING** modules. As well, certain features of level two in these modules have been supported. These extra language elements include full support for the **PERFORM, STRING,** and **UNSTRING** verbs. No support is provided for tape hardware. Some of the features described in the manual may not be present in specific hardware/software environments which do not provide adequate support for them. The chapter describing System Dependencies should be consulted for the specific details which apply to a particular implementation of Waterloo microCOBOL.

This reference manual describes the language supported by Waterloo microCOBOL. It is intended to be used for reference, *not* as a primer or tutorial. Waterloo microCOBOL is implemented on a number of different computer systems. Most of the manual applies to all implementations. The chapter about System Dependencies describes features particular to a specific system.

The following conventions are used in the formal descriptions of COBOL syntax:

(1) All reserved words are capitalized. When the words are required within the context that they are used, they are also shown in bold face.

(2) Square brackets [and] are used to mark the optional parts of the language. In cases where a choice must be made between a number of elements, the elements are shown in a vertical list enclosed by curly braces { and }.

(3) Semicolon (;) and comma (,) characters are optional items in the formal descriptions of COBOL elements. To increase the readability of these descriptions they have not been enclosed in square parentheses.

It should be noted that comma (,) and semicolon (;) characters may be used interchangeably in the COBOL language. When used as separators, these characters should be followed by a space character.

2.2 Divisions

A COBOL program is written as a number of **DIVISIONS.** Waterloo microCOBOL supports four of these divisions:

(1) **IDENTIFICATION DIVISION**

(2) **ENVIRONMENT DIVISION**

(3) **DATA DIVISION**

(4) **PROCEDURE DIVISION**

The divisions must be given in the order indicated.

The **IDENTIFICATION DIVISION** contains statements which are used to identify the program and other elements. The **ENVIRONMENT DIVISION** contains sentences describing the environment in which the program is intended to execute. The **DATA DIVISION** is used to declare the data upon which the executing program will operate. The **PROCEDURE DIVISION** contains sentences which, when executed, cause specific actions to take place. These divisions are described in detail in following chapters.

All four divisions are mandatory and so must be present in every program. Thus, the format of a COBOL program is as follows:

```
IDENTIFICATION DIVISION.
. . . . sentences
ENVIRONMENT DIVISION.
. . . . sentences
DATA DIVISION.
. . . . sentences
PROCEDURE DIVISION.
. . . . sentences
```

The sentences in each division are described in the following chapters. A description of the format of a COBOL program is given in the next section.

2.3 Columns in a COBOL Program

The following columns of a line in a program are significant:

column (1)	This column may contain an asterisk (*) character to indicate that the line is a comment line. Comment lines are ignored during the execution of a program. Their purpose is only to provide documentation for people who are looking at the source lines of the COBOL program.
column (2-5)	This area is called *Area A*. Certain sentences or statements must start in this area (e.g., paragraph names in the **PROCEDURE DIVISION**).
columns (6-)	This area is called *Area B*. Certain sentences must start in this area (e.g., verbs in the **PROCEDURE DIVISION**).

All COBOL statements, other than comments, must start in Area A or B. In general, the majority of the statements occur in the **PROCEDURE DIVISION.** The rules to follow in this division are as follows:

(1) **SECTION** and **PARAGRAPH** names start in Area A.

(2) All other sentences start in Area B.

If anything other than an asterisk (*) is placed in column (1), the editor supplied with microCOBOL will replace that character with a question mark (?) character to show an illegal character.

2.4 COBOL NAMES

Within a COBOL program, a number of names can be specified. These names are either *reserved words* (i.e., **SELECT**, **PROCEDURE**) or are defined by the programmer (i.e., file names, data names). A complete list of the reserved words in COBOL is given as an appendix (see RESERVED WORDS).

The names defined by the programmer must contain only alphabetic (A-Z, a-z), numeric (0-9) or dash (-) characters. A name may not start or end with a dash (-) character. All names must contain at least one alphabetic character. Names may include up to 30 characters.

The following are examples of legal COBOL names:

GOOD-DATA
TRANSACTION-FILE
DATE-004
1-PARAGRAPH-CHARLIE

The following are examples of illegal user-defined names:

WRITE	(reserved word)
-A	(starts with-)
B-	(ends with-)
403-7	(no alphabetic)

When entering a COBOL program using the editor supplied with microCOBOL, the following conventions are observed.

(1) lower and uppercase letters are treated identically in names.

(2) names will be subsequently displayed by the editor using the case of the first letter.

Thus, the names entered as

MyVariable and mYvARIABLE

are treated as being the same name. They would be displayed as

MYVARIABLE and myvariable.

2.5 Comment Statements

Comment statements may be entered anywhere in a COBOL program. These statements are ignored during the execution of the program. Their use is restricted to increasing the readability of the program by permitting documentation to be placed with the source statements. A comment statement is identified by an asterisk (*) in the first column of a statement.

2.6 Figurative Constants

Certain reserved words, called *figurative constants*, are used to stand for one or more repetitions of certain characters. These constants are as follows:

- **ZERO**, **ZEROS**, **ZEROES**: one or more "0" characters.
- **SPACE**, **SPACES**: one or more space characters.
- **HIGH-VALUE**, **HIGH-VALUES**: one or more of the character that has the highest ordinal position in the program collating sequence.
- **LOW-VALUE**, **LOW-VALUES**: one or more of the character that has the lowest ordinal position in the program collating sequence.
- **QUOTE**, **QUOTES**: one or more of the quotation (") character.
- **ALL** literal: one or more of the string of characters comprising the literal. The literal must be nonnumeric or a figurative constant (in which case the **ALL** keyword is redundant).

The singular and plural forms of a figurative constant are equivalent and may be used interchangeably.

The size of a figurative constant depends upon the context in which it is used. When the constant is associated with a specific data item (e.g., moved, compared, **VALUE IS**), the size of the constant is identical to that of the data item; otherwise, the literal has a size of one character (e.g., **DISPLAY**ing a figurative constant).

A figurative constant may be used anywhere that a literal can be used. When only a numeric literal is permitted, then only the **ZERO**, **ZEROS** or **ZEROES** figurative constants are permitted.

Chapter 3

IDENTIFICATION DIVISION

3.1 Overview

The **IDENTIFICATION DIVISION** must be the first division in a COBOL program (it may be preceded by comment statements). The statement

IDENTIFICATION DIVISION.

specifies the start of the division. The statement must start in Area A.

The division is used to identify the program in a general fashion. The division consists of a number of paragraphs. Each paragraph starts with a paragraph header (a reserved word written in Area A). The remainder of the paragraph is written in Area B. Only the **PROGRAM-ID** paragraph is mandatory. The remainder may be omitted from a program. When present, the paragraphs must be given in the following order:

PROGRAM-ID

AUTHOR

INSTALLATION

DATE-WRITTEN

DATE-COMPILED

SECURITY

Except for the **PROGRAM-ID** paragraph, the contents of each paragraph is limited to a single line which may contain anything and is ignored. Essentially, these entries may be considered to be documentation. The following sections describe each paragraph.

3.2 PROGRAM-ID

PROGRAM-ID. name.

This paragraph is used to give a name to the program. This name is not used in the Waterloo microCOBOL interpreter. The name is used to create the name of the object file in the Waterloo microCOBOL compiler.

This is the only mandatory paragraph in the **IDENTIFICATION DIVISION.**

3.3 AUTHOR

[**AUTHOR.** [comment]]

This paragraph is intended for documentation purposes only.

3.4 INSTALLATION

[**INSTALLATION.** [comment]]

This paragraph is intended for documentation purposes only.

3.5 DATE-WRITTEN

[**DATE-WRITTEN.** [comment]]

This paragraph is intended for documentation purposes only.

3.6 DATE-COMPILED

[**DATE-COMPILED.** [comment]]

This paragraph is intended for documentation purposes only.

3.7 SECURITY

[**SECURITY.** [comment]]

This paragraph is intended for documentation purposes only.

Chapter 4

ENVIRONMENT DIVISION

4.1 Overview

This division is used to inform the COBOL system about the *environment* in which the COBOL program is to be processed. The **CONFIGURATION SECTION** is mandatory. In that section, only the **SOURCE-COMPUTER** and **OBJECT-COMPUTER** paragraphs are required. The **INPUT-OUTPUT** section must be specified only if files are used in the program.

The first statement,

ENVIRONMENT DIVISION.

specifies the start of the division. The statement must start in Area A. The sections and paragraphs start in Area A while the various clauses (except the **SELECT** statement) start in Area B.

4.2 CONFIGURATION SECTION

CONFIGURATION SECTION.

This mandatory section consists of a number of paragraphs to be described in the following sections. Only the **SOURCE-COMPUTER** and **OBJECT-COMPUTER** paragraphs are mandatory.

4.2.1 SOURCE-COMPUTER

SOURCE-COMPUTER. name [WITH **DEBUGGING MODE**].

This mandatory paragraph is intended to be used as documentation of the computer on which the COBOL program is compiled (compiler) or interpreted (interpreter). The Waterloo microCOBOL interpreter treats the entire paragraph as a comment.

4.2.2 OBJECT-COMPUTER

```
OBJECT-COMPUTER. name.

                             { WORDS      }
    [,MEMORY SIZE number     { CHARACTERS }]
                             { MODULES    }

    [,PROGRAM COLLATING SEQUENCE is name ]
```

This mandatory paragraph is intended to be used as documentation of the computer on which the COBOL program is executed. Waterloo microCOBOL treats the paragraph as a comment.

Waterloo microCOBOL uses only the native character set of the computer on which it is executed. The collating sequence (order of the characters) is that of the characters defined for the system in question (see SYSTEM DEPENDENCIES).

4.2.3 SPECIAL-NAMES

```
[ SPECIAL-NAMES.

      [,CURRENCY SIGN IS literal ]

      [,DECIMAL-POINT IS COMMA ] ].
```

This paragraph is used to specify the currency-sign and decimal-point characters.

The *currency symbol* is a character used in **PICTURE** strings. It is normally used to precede values of money which are to be displayed. This character is dollar-sign ($) by default.

Usually the *decimal-point character* is a period (.). This character has special significance in **PICTURE** strings, in combination with comma (,) characters. The roles of these two characters can be reversed by specifying the **DECIMAL-POINT** clause.

4.3 INPUT-OUTPUT Section

```
INPUT-OUTPUT SECTION.
```

This optional section is used to specify the names and characteristics of files in the program. Each file used in the program must have **SELECT** clause in the **FILE-CONTROL** paragraph.

4.3.1 FILE-CONTROL

```
FILE-CONTROL.

    {select clause}
```

This optional paragraph consists of a number of **SELECT** clauses, one per file in the program, which are used to specify the COBOL file-name for an actual file. The format of the **SELECT** entries are described in the following section.

4.3.1.1 SELECT Clause

```
SELECT [ OPTIONAL ] file-name

    ASSIGN TO literal

    [; ORGANIZATION IS { RELATIVE   }]
                       { SEQUENTIAL }

    [; ACCESS MODE IS  { SEQUENTIAL [,RELATIVE KEY IS name ]  }]
                       {                                      }
                       { {RANDOM  } , RELATIVE KEY IS name    }
                       { {DYNAMIC }                           }

    [; FILE STATUS IS name ].
```

There must be a **SELECT** entry for each file used in the program. The "file-name" identifier specifies the name by which the file will be referenced elsewhere in the program.

The **OPTIONAL** keyword is used to indicate that the file need *not* be present every time the program is executed; when processed as an input file, a non-existent file is treated as a file with no records. Thus, the **AT END** condition will be detected when the first **READ** statement is executed for that file (see **READ** statement). When the keyword is not given, a non-existent file used as input will cause an error message to be displayed and the execution of the program will be terminated. The **OPTIONAL** keyword may only be used with input files.

The mandatory **ASSIGN** clause may be used to specify the actual file to be processed by the COBOL program. The value of the literal in the **ASSIGN** clause is normally used as the name of the actual file. This name is the actual name of the file for the computer system in which the program will execute. It should be noted that this clause may be overridden by the **VALUE** clause of an **FD** entry in the **FILE SECTION** of the **DATA DIVISION** .

The optional **ORGANIZATION** clause is used to inform WATERLOO microCOBOL whether the file is organized with special characteristics for relative or sequential processing. On some systems (i.e., IBM VM/CMS) there is no special organization and the clause is treated as a comment, except for its effect upon the **ACCESS** clause.

The optional **ACCESS** clause specifies whether the file will be accessed sequentially, randomly or both ways. When the **ORGANIZATION** is given as **SEQUENTIAL,** the only **ACCESS MODE** permissible is also **SEQUENTIAL** and the **RELATIVE KEY** clause may not be specified.

When the **ORGANIZATION** is given as **RELATIVE** the access mode may be any of **SEQUENTIAL, RANDOM** or **DYNAMIC**. The last mode specifies that both random and sequential access may be used for the file in question. The **RELATIVE KEY** clause may be specified for **SEQUENTIAL** access and must be specified for **RANDOM** or **DYNAMIC** access. When the **RELATIVE KEY** clause is specified, the data item indicated by the clause receives the relative record number of the record when a **READ** statement is successfully executed; and, the contents of the data item are used to establish the position in the file at which a record is to be written using a **WRITE** or **REWRITE** statement.

The optional **FILE STATUS** clause is used to specify a two-character *alphanumeric* or group data item to receive a value indicating the status of the last input/output operation for a file. The first character of this data item receives the following information immediately after an input/output operation:

"0"	operation completed successfully
"1"	**AT END** error detected
"2"	**INVALID KEY** error detected
"3"	other input/output error

When an **INVALID KEY** error (value 2) is detected the second character of the **FILE STATUS** data item may contain the following values:

"2"	record already exists
"3"	no record found
"4"	attempt to access a record beyond the bounds of a file

Otherwise, the second character of the data item will contain "0".

Two typical **SELECT** statements are illustrated below:

```
select myfile
     assign to "TRANS".

select output-file
     assign to 'MASTER'
     organization is relative
     access mode is random
     relative key is master-rec-numb.
```

The **SELECT** statements show how a sequential and a relative file, respectively, might be referenced. The relative file is to be accessed in a **RANDOM** mode.

Chapter 5

DATA DIVISION

5.1 Overview

The **DATA DIVISION** is used to inform Waterloo microCOBOL about the data used in the program. There are two sections which deal with this data: **FILE SECTION** (input/output records) and **WORKING-STORAGE SECTION** (other data items used in the program). These COBOL **SECTION**s are described in detail in the following sections.

The first statement in the division is

DATA DIVISION.

It must start in Area A. Data items are all preceded by level numbers (described in the next section). Level numbers 01 and 77 must start in Area A. All other level numbers may be indented arbitrarily. The data names following the level numbers must start in Area B.

5.2 FILE SECTION

FILE SECTION.

This optional section is concerned with the data that applies to the files used in a program. For each file there is an **FD** (file description) entry specified. The **FD** is used to describe the information in the file. Each **FD** is immediately followed by one or more record descriptions which define the format of a record read from or to be written to a file.

5.2.1 FD

```
FD filename

(FD entry)

(record-description entry) . . . ]
```

where an **FD** entry is described as:

```
[; BLOCK contains [ number TO ] number      { RECORDS    }]
                                            { CHARACTERS }

[; RECORD CONTAINS [ number TO ] number CHARACTERS ]

[;LABEL {RECORD IS    } { STANDARD  } ]
        {RECORDS ARE  } { OMITTED   }

[; VALUE OF literal is literal

[;DATA { RECORD IS   } name, name ... ]
       { RECORDS ARE }

[; CODE-SET IS name ]
```

An **FD** entry must be present for each file used in the COBOL program. There is one mandatory clause, **LABEL**, and a number of optional clauses. These clauses may be given in any order. These clauses are described in detail in the following subsections.

A typical **FD** entry is shown following:

```
fd  myfile
    label records are standard.
01  my-data.
    02 filler pic x(50).
```

The file contains 50-character records.

5.2.1.1 BLOCK CONTAINS

The optional **BLOCK** clause is used to inform microCOBOL of the size of a physical block in which logical records are stored. In many implementations (i.e., SuperPET, VM/CMS) this information is not required and so is ignored except for syntactic correctness.

The clause specifies the size of a physical block either in records or in characters.

5.2.1.2 RECORD CONTAINS

This optional clause is never required since the size of logical records is also be given by the record description entries following the **FD**. When specified, the size or range of sizes must be the same as that of the single record, or as the range of sizes given by those multiple records, respectively.

5.2.1.3 LABEL

This mandatory clause must be present in each **FD**. It is used to indicate whether a special record, called a label record, precedes the data records in a file. Because many specific systems (i.e., SuperPET, VM/CMS) automatically handle label records, this clause is usually ignored, except for syntactic correctness.

5.2.1.4 VALUE OF

This optional clause may be used to specify the system name of the file in question. When the value is given as a literal, its value is taken as the system name for the file. Thus, the clause

```
VALUE OF '' IS 'SALES'
```

indicates that the system name for the file is "SALES".

When the value is given as a data name, the value of that data item (at the time the file is opened) is used as the system name. It should be noted that this feature allows the name of a file to be established while the COBOL program is being executed. For example, the name may be entered by a user of a program via an **ACCEPT** statement.

The clause

```
VALUE OF '' IS FILENAME
```

specifies that the value of the data item "FILENAME" is to be used as the system name for the file, when the file is opened.

5.2.1.5 DATA

This optional clause is used only to document the records given immediately following the **FD** entry. If specified, the names of the records must be the same as the 01 level record descriptions following the **FD.**

5.2.1.6 CODE SET

The CODE SET clause is used to specify the character set to be used for the data in the records in the file. The only character set supported by microCOBOL is the one that is normally used on the system, called **NATIVE.** Thus, the only valid form of this clause is

```
CODE SET IS NATIVE.
```

5.2.2 Record Descriptions

The record descriptions following an **FD** establish the size(s) of logical record(s) written to or read from a file. The format of record descriptions is given in the section dealing with data description.

5.3 WORKING-STORAGE SECTION

```
[ WORKING-STORAGE SECTION.

{ 77 (data-description)            } . . . ]
{ (record-description entry)       }
```

Data in the **WORKING-STORAGE SECTION** is defined using either 77 level entries (elementary items) or 01 level entries (record descriptions). These entries are described in the next section.

5.4 Data Description

This section is used to specify the data items (other than file records) used by the COBOL program. The various entries are given in following subsections.

5.4.1 Level Numbers and Records

Each data item is given a level number in order to organize *elementary items* as subdivisions of *group items*. An item which is a subdivision of another item is said to be *subordinate* to all the group items which contain it. In general, the specification of a larger level number indicates that the data item specified is a subdivision of the data item with a lesser level number that immediately precedes it in the program. This may be schematically shown as follows:

```
01 A
    02 B
    02 C
        03 D
        03 E
    02 F
        05 G
        05 H
```

The group item A is subdivided into 3 items shown as B, C and F. The group item C is subdivided into D and E. The group item F is subdivided into items G and H. Items which are not subdivided (B, D, E, G, H) are termed *elementary items*. It is important to differentiate between group and elementary items since different clauses can be used to describe the data contained by them.

A group item with a level number 01 is called a *record*. When subdividing records or group items in records, the following rules must be followed:

(1) All items which are used to subdivide a group item must have the same level number.

(2) Level numbers used in records must be in the range 01 to 49.

Records defined in this way are used in both the **WORKING-STORAGE** and **FILE** sections.

There are also 3 reserved level numbers (66,77,88) used for special purposes. Briefly, these purposes are:

66	used to regroup data items
77	used to define special elementary items
88	used to define condition names

These items are described in detail in following subsections.

5.4.2 Qualification

```
name { OF } name [  { OF } name ] ...
     { IN }         { IN }
```

Any data name which is used as a subdivision of a group item may be qualified by specifying some or all of the names of group items of which it is part. Consider the following schematic diagram:

```
01 A
   02 B
      03 C
         04 D
....
```

The data item D may be referenced elsewhere in the program by any of the following:

```
D
D OF C
D OF B
D OF C OF B
D OF A
D OF C OF A
D OF B OF A
D OF C OF B OF A
```

Qualification must be used when a name occurs more than once in the same program

Consider the following example:

```
01  A
      02   B
 ....
01  C
      02  D
            03  B
```

In the example, B cannot be used by itself since the reference would be ambiguous. The first occurance of B must be referenced as

```
B OF A
```

and the second occurrence must be referenced by one of:

```
B OF D
B OF C
B OF D OF C
```

Every data item must be capable of being uniquely qualified.

5.4.3 PICTURE Strings

A *PICTURE string* is a sequence of characters (maximum 30) which is used to describe all *elementary* data items (except **INDEXED**). A few examples of **PICTURE** strings are as follows:

PICTURE	Meaning
999V99	5-digit number with 2 decimal places
XXXXXX	6-character sequence of characters
ZZZZ9	5-digit field, leading zeros suppressed on 4 digits

Uppercase letters will be used as the characters in **PICTURE** strings for explanatory purposes. Waterloo microCOBOL also accepts lowercase letters in an equivalent fashion. The general rules for **PICTURE** strings are as follows:

(1) There are five categories of data that can be described with a **PICTURE** clause: alphabetic, numeric, alphanumeric, alphanumeric edited, and numeric edited.

(2) To define an item as *alphabetic*:

a. Its **PICTURE** character-string can only contain the symbols 'A', 'B'; and

b. Its contents when represented in standard data format must be any combination of the twenty-six (26) letters of the Roman alphabet and the space from the COBOL character set.

Some sample alphabetic **PICTURE** strings are shown following:

```
A(20)
bbbb
a(10)BBAA
```

(3) To define an item as *numeric*:

a. Its **PICTURE** character-string can only contain the symbols '9', 'P', 'S', and 'V'. The number of digit positions that can be described by the **PICTURE** character-string must range from 1 to 18 inclusive; and

b. If unsigned, its contents when represented in standard data format must be a combination of the Arabic numerals '0', '1', '2', '3', '4', '5', '6', '7', '8', and '9'; if signed, the item may also contain a '+', '-', or other representation of an operational sign.

Some sample numeric strings are as follows:

```
99999
9999v99
S9999V9999
S9(6)v99
```

(4) To define an item as *alphanumeric*:

a. Its **PICTURE** character-string is restricted to certain combinations of the symbols 'A', 'X', '9', and the item is treated as if the character-string contained all X's. A **PICTURE** character-string which contains all A's or all 9's does not define an alphanumeric item; and

b. Its contents when represented in standard data format are allowable characters in the computer's character set.

Some sample alphanumeric strings are as follows:

```
999xxx
A(10)99X(4)
```

(5) To define an item as *alphanumeric edited*:

a. Its **PICTURE** character-string is restricted to certain combinations of the following symbols: 'A', 'X', '9', 'B', '0', and '/'; and

1) The character-string must contain at least one 'B' and at least one 'X' or at least one '0' (zero) and at least one 'X' or at least one '/' (stroke) and at least one 'X'; or

2) The character-string must contain at least one '0' (zero) and at least one 'A' or at least one '/' (stroke) and at least one 'A'; and

b. The contents when represented in standard data format are allowable characters in the computer's character set.

A sample alphanumeric string is:

BAA/AA/AAB

(6) To define an item as *numeric edited*:

a. Its **PICTURE** character-string is restricted to certain combinations of the symbols 'B', '/', 'P', 'V', 'Z', '0', '9', ',', '.', '*', '+', '-', 'CR', 'DB', and the currency symbol. The allowable combinations are determined from the order of precedence of symbols and the editing rules; and

1) The number of digit positions that can be represented in the **PICTURE** character-string must range from 1 to 18 inclusive; and

2) The character-string must contain at least one '0', 'B', '/', 'Z', '*', '+', ',' '.', '-', 'CR', 'DB', or currency symbol.

b. The contents of the character positions of these symbols that are allowed to represent a digit in standard data format, must be one of the numerals.

Some sample numeric edited strings are:

99/99/99
$$,$$$,$$9.99
,9.99
zzz,zzz,zz9.99
$$$,$$$,$$9.99CR

(7) The *size of an elementary item*, where size means the number of character positions occupied by the elementary item in standard data format, is determined by the number of allowable symbols that represent character positions. An integer which is enclosed in parentheses following the symbols 'A', ',', 'X', '9', 'P', 'Z', '*', 'B', '/', '0', '+', '-', or the currency symbol indicates the number of consecutive occurrences of the symbol. Note that the following symbols may appear only once in a given **PICTURE**: 'S', 'V', '.', 'CR', and 'DB'.

(8) The *functions of the symbols* used to describe an elementary item are explained as follows:

A Each 'A' in the character-string represents a character position which can contain only a letter of the alphabet or a space.

B Each 'B' in the character-string represents a character position into which the space character will be inserted.

P Each 'P' indicates an assumed decimal scaling position and is used to specify the location of an assumed decimal point when the point is not within the number that appears in the data item. The scaling position character 'P' is not counted in the size of the data item. Scaling position characters are counted in determining the maximum number of digit positions (18) in numeric edited items or numeric items. The scaling position character 'P' can appear only to the left or right as a continuous string of 'P's within a **PICTURE** description; since the scaling position character 'P' implies an assumed decimal point (to the left of 'P's if 'P's are leftmost **PICTURE** characters and to the right if 'P's are rightmost **PICTURE** characters), the assumed decimal point symbol 'V' is redundant as either the leftmost or rightmost character within such a **PICTURE** description. The character 'P' and the insertion character '.' (period) cannot both occur in the same **PICTURE** character-string. If, in any operation involving conversion of data from one form of internal representation to another, the data item being converted is described with the **PICTURE** character 'P', each digit position described by a 'P' is considered to contain the value zero, and the size of the data item is considered to include the digit positions so described.

S The letter 'S' is used in a character-string to indicate the presence, but neither the representation nor, necessarily, the position of an operational sign; it must be written as the leftmost character in the **PICTURE.** The 'S' is not counted in determining the size (in terms of standard data format characters) of the elementary item unless the entry is subject to a SIGN clause which specifies the optional **SEPARATE CHARACTER** phrase.

V The 'V' is used in a character-string to indicate the location of the assumed decimal point and may only appear once in a character-string. The 'V' does not represent a character position and therefore is not counted in the size of the elementary item. When the assumed decimal point is to the right of the rightmost symbol in the string the 'V' is redundant.

X Each 'X' in the character-string is used to represent a character position which contains any allowable character from the computer's character set.

Z Each 'Z' in a character-string may only be used to represent the leftmost leading numeric character positions which will be replaced by a space character when the contents of that character position is zero. Each 'Z' is counted in the size of the item.

9 Each '9' in the character-string represents a character position which contains a numeral and is counted in the size of the item.

0 Each '0' (zero) in the character-string represents a character position into which the numeral zero will be inserted. The '0' is counted in the size of the item.

/ Each '/' (stroke) in the character-string represents a character position into which the stroke character will be inserted. The '/' is counted in the size of the item.

, Each ',' (comma) in the character-string represents a character position into which the character ',' will be inserted. This character position is counted in the size of the item. The insertion character ',' must not be the last character in the **PICTURE** character-string.
When the character '.' (period) appears in the character-string it is an editing symbol which represents the decimal point for alignment purposes and in addition, represents a character position into which the character '.' will be inserted. The character '.' is counted in the size of the item.

For a given program the functions of the period and comma are exchanged if the clause **DECIMAL-POINT IS COMMA** is stated in the **SPECIAL-NAMES** paragraph. In this exchange the rules for the period apply to the comma and the rules for the comma apply to the period wherever they appear in a **PICTURE** clause. The insertion character '.' must not be the last character in the **PICTURE** character-string.

+, −, CR, DB

These symbols are used as editing sign control symbols. When used, they represent the character position into which the editing sign control symbol will be placed. The symbols are mutually exclusive in any one character-string and each character used in the symbol is counted in determining the size of the data item.

*	Each '*' (asterisk) in the character-string represents a leading numeric character position into which an asterisk will be placed when the contents of that position is zero. Each '*' is counted in the size of the item.
cs	The currency symbol in the character-string represents a character position into which a currency symbol is to be placed. The currency symbol in a character-string is represented by either the currency sign$ or by the single character specified in the **CURRENCY SIGN** clause in the **SPECIAL-NAMES** paragraph. The currency symbol is counted in the size of the item.

When a value is assigned to a data item, it is said to be *edited* into that item. The following rules describe this process.

(1) There are two general methods of performing editing in the **PICTURE** clause, either by insertion or by suppression and replacement. There are four types of insertion editing available. They are:

a. Simple insertion

b. Special insertion

c. Fixed insertion

d. Floating insertion

There are two types of suppression and replacement editing:

a. Zero suppression and replacement with spaces

b. Zero suppression and replacement with asterisks

(2) The type of editing which may be performed upon an item is dependent upon the category to which the item belongs. The following table specifies which type of editing may be performed upon a given category:

CATEGORY	TYPE OF EDITING
Alphabetic	Simple insertion 'B' only
Numeric	None
Alphanumeric	None
Alphanumeric Edited	Simple Insection '0', 'B', and '/'
Numeric Edited	All, see rule 3 following

(3) Floating insertion editing and editing by zero suppression and replacement are mutually exclusive in a **PICTURE** clause. Only one type of replacement may be used with zero suppression in a **PICTURE** clause.

(4) *Simple Insertion Editing*. The ',' (comma), 'B' (space), '0' (zero), and '/' (stroke) are used as the insertion characters. The insertion characters are counted in the size of the item and represent the position in the item into which the character will be inserted.

(5) *Special Insertion Editing*. The '.' (period) is used as the insertion character. In addition to being an insertion character it also represents the decimal point for alignment purposes. The insertion character used for the actual decimal point is counted in the size of the item. The use of the assumed decimal point, represented by the symbol 'V' and the actual decimal point, represented by the insertion character, in the same **PICTURE** character-string is disallowed. The result of special insertion editing is the appearance of the insertion character in the item in the same position as shown in the character-string.

(6) *Fixed Insertion Editing*. The currency symbol and the editing sign control symbols, '+', '-', 'CR', 'DB', are the insertion characters. Only one currency symbol and only one of the editing sign control symbols can be used in a given **PICTURE** character-string. When the symbols 'CR' or 'DB' are used they represent two character positions in determining the size of the item and they must represent the rightmost character positions that are counted in the size of the item. The symbol '+' or '-', when used, must be either the leftmost or rightmost character position to be counted in the size of the item. The currency symbol must be the leftmost character position to be counted in the size of the item except that it can be preceded by either a '+' or a '-' symbol. Fixed insertion editing results in the insertion character occupying the same character position in the edited item as it occupied in the **PICTURE** character-string. Editing sign control symbols produce the following results depending upon the value of the data item:

EDITING SYMBOL	DATA NON-NEGATIVE	DATA NEGATIVE
+	+	-
-	space	-
CR	2 spaces	CR
DB	2 spaces	DB

(7) *Floating Insertion Editing*. The currency symbol and editing sign control symbols '+' or '-' are the floating insertion characters and as such are mutually exclusive in a given **PICTURE** character-string.

Floating insertion editing is indicated in a **PICTURE** character-string by using a string of at least two of the floating insertion characters. This string of floating insertion characters may contain any of the fixed insertion symbols or have fixed insertion characters immediately to the right of this string. These simple insertion characters are part of the floating string.

The leftmost character of the floating insertion string represents the leftmost limit of the floating symbol in the data item. The rightmost character of the floating string represents the rightmost limit of the floating symbols in the data item.

The second floating character from the left represents the leftmost limit of the numeric data that can be stored in the data item. Non-zero numeric data may replace all the characters at or to the right of this limit.

In a **PICTURE** character-string, there are only two ways to representing floating insertion editing. One way is to represent any or all of the leading numeric character positions on the left of the decimal point by the insertion character. The other way is to represent all of the numeric character positions in the **PICTURE** character-string by the insertion character.

If the insertion characters are only to the left of the decimal point in the **PICTURE** character-string, the result is that a single floating insertion character will be placed into the character position immediately preceding either the decimal point or the first non-zero digit in the data represented by the insertion symbol string, whichever is farther to the left in the **PICTURE** character-string. The character positions preceding the insertion character are replaced with spaces.

If all numeric character positions in the **PICTURE** character-string are represented by the insertion character, the result depends upon the value of the data. If the value is zero the entire data item will contain spaces. If the value is not zero, the result is the same as when the insertion character is only to the left of the decimal point.

To avoid truncation, the minimum size of the **PICTURE** character-string for the receiving data item must be the number of characters in the sending data item, plus the number of non-floating insertion characters being edited into the receiving data item, plus one for the floating insertion character.

(8) *Zero Suppression Editing*. The suppression of leading zeroes in numeric character positions is indicated by the use of the alphabetic character 'Z' or the character '*' (asterisk) as suppression symbols in a **PICTURE** character-string. These symbols are mutually exclusive in a given **PICTURE** character-string. Each suppression symbol is counted in determining the size of the item. If 'Z' is used the replacement character will be the space and if the asterisk is used, the replacement character will be '*'.

Zero suppression and replacement is indicated in a **PICTURE** character-string by using a string of one or more of the allowable symbols to represent leading numeric character positions which are to be replaced when the associated character position in the data contains a zero. Any of the simple insertion characters embedded in the string of symbols or to the immediate right of this string are part of the string.

In a **PICTURE** character-string, there are only two ways of representing zero suppression. One way is to represent any or all of the leading numeric character positions to the left of the decimal point by suppression symbols. The other way is to represent all of the numeric character positions in the **PICTURE** character-string by suppression symbols.

If the suppression symbols appear only to the left of the decimal point, any leading zero in the data which corresponds to a symbol in the string is replaced by the replacement character. Suppression terminates at the first non-zero digit in the data represented by the suppression symbol string or at the decimal point, whichever is encountered first.

If all numeric character positions in the **PICTURE** character-string are represented by suppression symbols and the value of the data is not zero the result is the same as if the suppression characters were only to the left of the decimal point. If the value is zero and the suppression symbol is 'Z', the entire data item will be spaces. If the value is zero and the suppression symbol is '*', the data item will be all '*' except for the actual decimal point.

(9) The symbols '+', '-', '*', 'Z', and the currency symbol, when used as floating replacement characters, are mutually exclusive within a given character-string.

The following chart shows legal combinations of picture characters. An "x" at an intersection indicates that the symbol at the top of a column may precede the symbol at the left of a row. The currency symbol is indicated by a dollar sign ($) character.

		FIXED							FLOAT						OTHER					
		B 0 /	,	.	+ -	+ -	CR DR	$	Z *	Z *	+ -	+ -	$	$	9	A X	S	V	P	P
F	B 0 /	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
I	,	x	x	x	x			x	x	x	x	x	x	x	x			x		x
X	.	x	x		x			x	x		x		x		x					
E	+ -																			
D	+ -	x	x	x				x	x	x			x	x	x			x	x	x
	CR DR	x	x	x				x	x	x			x	x	x			x	x	x
	$				x															
F	Z	x	x		x			x	x											
L	Z	x	x	x	x			x	x	x								x		x
O	+ -	x	x					x			x									
A	+ -	x	x	x				x			x	x						x		x
T	$	x	x		x								x							
	$	x	x	x	x								x	x				x		x
O	9	x	x	x	x			x	x		x		x		x	x	x	x		x
T	A X	x													x	x				
H	S																			
E	V	x	x		x			x	x		x		x		x		x		x	
R	P	x	x		x			x	x		x		x		x		x		x	
	P				x			x									x	x		x

Non-floating insertion symbols '+' and '-', floating insertion symbols 'Z', '*', '+', '-', '$', and the other symbol 'P' appear twice in the preceding chart. The leftmost column and uppermost row for each symbol represent its use to the left of the decimal point position. The second appearance of the symbol in the chart represents its use to the right of the decimal point.

The following characters are mutually exclusive in a **PICTURE** string:

non-floating '+' and '-'
'CR' and 'DB'
'Z' and '*'
floating '+' and '-'

At least one of the symbols 'A', 'X', 'Z', '9' or '*', or at least two of the characters '+', '-' or '$' must be present in a **PICTURE** string.

5.4.4 Describing Data Items

```
level-number   { data-name  }
               { FILLER     }

     [; REDEFINES data-name ]

     [; { PICTURE  } IS character string ]
        {PIC       }

                     { COMPUTATIONAL  }
     [; [ USAGE IS ]  { COMP           } ]
                     { DISPLAY        }
                     { INDEX          }

     [; SIGN IS ]   { LEADING  } [ SEPARATE CHARACTER ]
                    {TRAILING }

     [; OCCURS  { number TO number TIMES DEPENDING on name } ]
                { number TIMES }

         [ INDEXED BY name [, name ....] ]

     [;  { SYNCHRONIZED  } [  { LEFT   } ] ]
         { SYNCH         }    { RIGHT  }

     [;   { JUSTIFIED  } RIGHT ]
          { JUST       }

     [; BLANK WHEN ZERO ]

     [; VALUE is literal ] .
```

This section indicates how data items may be described. Items with level numbers 66 or 88 are described in the next section. The following general rules apply:

(1) the clauses (described in subsections) may be written in any order except

  (a) the data name or **FILLER** keyword must immediately follow the level number; and

  (b) when the **REDEFINES** clause is used, it must immediately follow the data name or **FILLER** keyword.

(2) A **PICTURE** clause must be specified for every elementary item except for an index data item in which case the clause may not be used.

(3) A 77 level data item must be specified as an elementary data item. The **FILLER** keyword cannot be used with this level number.

(4) The following keywords are equivalent:

THRU	THROUGH
PIC	PICTURE
COMP	COMPUTATIONAL
SYNC	SYNCHRONIZED
JUST	JUSTIFIED

(5) The **SYNCHRONIZED**, **PICTURE**, **JUSTIFIED**, and **BLANK WHEN ZERO** clauses may be used only with elementary data items.

The following subsections describe the various clauses.

The **FILLER** keyword may be used instead of a data name when the elementary item is never to be explicitly referenced. Thus, it is used to reserve storage which will be referenced in some other manner such as using the group item containing it. The **FILLER** keyword may be used many times in a program.

5.4.4.1 BLANK WHEN ZERO

This clause is used to indicate that the item is to contain spaces when its value is zero. It may only be specified for an elementary item which is numeric or numeric edited. The category of an item containing this clause is considered to be numeric edited.

5.4.4.2 JUSTIFIED

This clause is used to specify non-standard positioning of data within a data item. It may only be specified for an elementary item which is neither numeric nor edited.

Normally, when data is moved to a field it is moved as follows:

(1) When the data is larger than the field, the data is truncated on the right and placed in the field.

(2) When the data is smaller than the field, the data is augmented with space characters on the right and placed in the field.

The **JUSTIFIED** clause changes this normal action as follows:

(1) When the data is larger than the field, the data is truncated on the left and placed in the field.

(2) When the data is smaller than the field, the data is augmented with space characters on the left and placed in the field.

In both cases, the rightmost position of the data is placed in the rightmost position of the receiving field.

5.4.4.3 OCCURS Clause

The **OCCURS** clause is described in the chapter entitled TABLE HANDLING.

5.4.4.4 PICTURE Clause

The **PICTURE** clause is used to describe the general characteristics and editing requirements of an elementary data item. This is accomplished by the **PICTURE** string following the **PICTURE** keyword (see **PICTURE STRING**). The clause may only be specified for elementary data items.

5.4.4.5 REDEFINES

The **REDEFINES** clause is used to provide another definition of a previously defined area of storage. The two data names must have the same level number and there must be no data item with a lower level number between these two data items in the program. Level 66 and 88 items may not use **REDEFINES**.

The clause is used in order to provide more than one definition of how an area of storage is to be treated. This enables, for example, two or more **PICTURE** clauses to apply to a single area of storage.

The following rules apply to the data item following the **REDEFINES** keyword:

(1) The data item may not itself contain a **REDEFINES** clause, although it may be subordinate to a group item which does contain a **REDEFINES** clause. Thus, the following is illegal:

```
05 A.
. . .
05 B REDEFINES A.
. . .
05 C REDEFINES B.
```

The data item "C" is **REDEFINE**d using the data item "C" which is itself **REDEFINE**d. The following is legal:

```
03 X REDEFINES Y.
. . .
    05 A.
    . . .
    05 B REDEFINES A.
    . . .
    05 C REDEFINES A.
```

(2) The data item cannot contain an **OCCURS** clause. The following would be illegal:

```
10 TAB-COST OCCURS 20 TIMES.
. . .
10 TAB REDEFINES TAB-COST.
```

The following rules apply to the data item following the level number:

(1) The data item may not use an **OCCURS** clause.

(2) The data item, or any items subordinate to it, may not contain a **VALUE** clause, except for condition name entries. The **VALUE** clause can only be used with an item which actually defines storage. Any item which is **REDEFINE**d to occupy existing storage cannot have this clause. Thus,

```
10  A  PIC 99 VALUE IS 47.
. . .
10  B  REDEFINES A PIC XX.
```

is legal, while

```
10  A  PIC 99.
. . .
10  B  REDEFINES A PIC XX VALUE IS "99".
```

is illegal.

The following rules apply to both data items:

(1) The items may not have 01 level numbers in the **FILE SECTION**.

(2) When the items do not have 01 level numbers, they must be the same size.

5.4.4.6 SIGN

This clause is used to indicate the position and representation of signs of numeric data items. It may be used only with numeric data items with a **PICTURE** string containing an "S" or with group items which contain at least one such data item.

When the clause has not been specified for either an elementary numeric data item whose picture contains an "S" or for a group item containing it, the sign is stored in the same storage location as the right-most character of the data item. Because this last character is used to store both the last digit and the sign, an attempt to **DISPLAY** the last character will cause it to appear as another character than the digit. The data may be reviewed in a more understandable format by moving the data to a numeric edited data item and then **DISPLAY**ing that item.

The **SIGN** clause is used to obtain the storage of the sign information in different ways:

(1) When **TRAILING** is specified and **SEPARATE** is present, the sign is stored as a "+" or a "-" in a character following the last digit.

(2) When **LEADING** is specified and **SEPARATE** is present, the sign is stored as a "+" or a "-" in a character preceding the first digit.

(3) When **TRAILING** is specified and **SEPARATE** is not given, the default representation of the sign (described previously) is used.

(4) When **LEADING** is specified and **SEPARATE** is not given, the default representation of the sign (described previously) is used except that the first digit of the data item contains the sign (not the last one).

5.4.4.7 SYNCHRONIZED

This clause is intended to be used to align data on the "natural" boundaries of storage that are required in some computer systems. In the systems in which microCOBOL is currently implemented or being implemented (IBM 370, DEC PDP/11, IBM Personal Computer, Motoral 6809) this alignment is unnecessary and so this clause has no effect. The clause may only be used with elementary data items.

5.4.4.8 USAGE

This clause is intended to provide different representations of data for items, depending upon whether they are used in computations or not. Waterloo microCOBOL uses the same representation for both **COMPUTATIONAL** and **DISPLAY** items.

An item specified as **COMPUTATIONAL** may have a **PICTURE** string containing only the characters "9", "S", "V" and 'P'. When the **USAGE** clause is specified for a group items it applies to the elementary items subordinate to it.

An item specified as **INDEX** can only be used to index items in tables (see INDEXING). The **SYNCHRONIZED**, **JUSTIFIED**, **PICTURE**, **VALUE**, and **BLANK WHEN ZERO** clauses cannot be used for an indexed data item.

5.4.4.9 VALUE

The **VALUE** clause is used to place an *initial value* in a data item. This is the value contained in the item when the program *begins* execution. When the storage for a data item has not been initialized in this way, the data item should not be used as value until a value has placed in the data item. The Waterloo microCOBOL Interpreter treats as an error any attempt to use such an undefined value.

The data value to be placed into the storage for a data item is specified as either a literal or a figurative constant. The rules for numeric and non-numeric items are given in the following paragraphs.

A numeric literal or a figurative constant may be specified as initialization for a numeric data item. A numeric literal must not be larger than the capability of the item to store that value. If the literal specifies a sign, the data item must be a signed numeric item.

Non-numeric data items, including group items, may be initialized with non-numeric literals or figurative constants. The size of the literal cannot exceed the size of the data item. No editing is performed; a literal is presented in an edited form.

The **JUSTIFIED** and **BLANK WHEN ZERO** clauses are ignored when the data is placed into data item as a result of the **VALUE** clause. The clause may not be used in the following cases:

(1) when the data item also contains an **OCCURS** or **REDEFINES** clause, or is subordinate to a group item containing those clauses;

(2) when the data item is in the **FILE SECTION**; or

(3) when the data item is group item which has subordinate items with any of the **JUSTIFIED, SYNCHRONIZED**, or **USAGE** (other than **DISPLAY**) clauses.

5.4.5 66 Level Data Items

```
66 name-1; RENAMES name-2 [  { THROUGH  } name-3 ]
                             { THRU     }
```

A 66 level data item defines an alternative method to group one or more elementary data items. The 66 level item is considered to be a group item, unless it renames a single elementary data item. The storage for the 66 level item begins at the start of the data item given following the **RENAMES** keyword and continues to the end of the data item specified in the **THRU** clause. When no **THRU** or **THROUGH** clauses are present, the storage ends at the end of the data item given following the **RENAMES** clause.

Consider the following example:

```
01 A.
   05 B.
      10 C PIC X.
      10 D PIC X.
   05 E.
      10 F PIC X.
      10 G PIC X.
      10 H PIC X.
66 DEF-1 RENAMES D THROUGH G.
66 DEF-2 RENAMES D THRU E.
```

The data item "DEF-1" includes the elementary items "D", "F", and "G"; the data item "DEF-2" includes the elementary items "D", "F" "G" and "H".

The following rules apply to 66 level data items:

(1) All 66 level items must occur immediately following the last data description in a logical record.

(2) The one or two data items specified in the **RENAMES** clause must be distinct items in the logical in which the 66 level item applies. These items may not be 01, 66, 77 or 88 level items. These items may not have an **OCCURS** clause nor can either be subordinate to an item with this clause.

(3) When the **THRU** or **THROUGH** keyword is used, the data item following the keyword must not start before and must end after the data item given following the **RENAMES** keyword.

(4) The 66 level data item is treated as an elementary item if the **THRU** clause is not used and if the data item following the **RENAMES** keyword is an elementary item; otherwise, the item is treated as a group item.

5.4.6 88 Level Data Items

```
88 name; { VALUE IS    } literal      [ { THROUGH  } literal ]
         { VALUES ARE  }              [ { THRU     }

         [ , literal [      { THROUGH  } literal ] ] . . .
                            { THRU     }
```

Level 88 data items are used to specify *condition names* to be associated with fields, called *condition variables*. Consider the following example:

```
77 TR-CODE PICTURE 99.
88 GOOD-CODE VALUES ARE 20, 30, 40.
88 ADD-CODE VALUE IS 20.
88 DLT-CODE VALUE IS 30.
88 CHG-CODE VALUE IS 40.
```

Four condition names are defined for the condition variable "TR-CODE". The usage of one of these 88 level items as a simple condition will result in a value of 'true' when the condition variable contains one of the values given with that 88 level item. Thus, the statement

```
IF GOOD-CODE
```

may be used to test if "TR-CODE" contains 20, 30 or 40. Similarly,

```
IF CHG-CODE
```

may be used to test if "TR-CODE" contains 40.

The 88 level items are given following the field to be used as the condition variable. The condition variable may not have any of the following properties:

(1) level 66

(2) **USAGE** given as **COMP**, **COMPUTATIONAL** or **INDEX** .

(3) **JUSTIFIED** or **SYNCHRONIZED** clauses.

The conditional variable may be an item in a logical record.

The values to be used to test if the condition name is true are given as single literals or as ranges of values (when **THRU** or **THROUGH** keywords used). The test to see if the conditional variable contains an appropriate value is equivalent to one of the following:

literal	condition-variable = literal
range	condition-variable NOT < first-literal AND condition-variable NOT > second-literal

where the literals used in the range test are respectively the literals before and after the **THRU** or **THROUGH** keyword.

Chapter 6

PROCEDURE DIVISION

6.1 Overview

```
    PROCEDURE DIVISION.
[   DECLARATIVES.
{   section-name SECTION. declarative sentence
[   paragraph-name. [ sentence ] ... ] } ...
    END DECLARATIVES. ]
    (procedure body)
```

where the procedure body is given by:

```
{   paragraph-name. [ sentence ] . . . }
```

or

```
{   section-name SECTION.
[   paragraph-name. [ sentence ] ... ] ... } ...
```

The **PROCEDURE DIVISION** is concerned with the actions to be performed by the program. The division consists of a number of paragraphs which consist of sentences to specify particular actions and of directives to specify actions to take place when certain error situations arise.

Each *paragraph* consists of zero or more sentences. Each *sentence* consists of one or more statements, followed by a period (.) character. Each *statement* starts with a verb. These verbs are discussed in the following chapters.

A number of the verbs are said to be *conditional* statements; i.e., they have a portion of them which is executed only if some condition is true. For example,

```
READ IN-FILE INTO IN-RECORD
     AT END MOVE HIGH-VALUES TO IN-KEY.
```

illustrates a **READ** statement in which a **MOVE** verb is executed when an attempt is made to read past the end of the file. When there is no conditional action associated with the statement, the statement is said to be an *imperative* statement. A sequence of imperative statements is also treated as an imperative statement. Many of the conditional statements specify that an imperative statement must be the conditional part of the conditional statement.

The optional **DECLARATIVES** portion occurs first in the **PROCEDURE DIVISION**. It is composed of a number of sections, each one of which has a **USE** statement that specifies an error condition. The section in question is executed whenever the specified error condition arises during the normal execution of the program. The **END DECLARATIVES** statement marks the end of the declaratives area.

The main body of the **PROCEDURE DIVISION** follows the optional **DECLARATIVES** area. The body consists either of a number of paragraphs or a number of sections each of which consists of a number of paragraphs. Section and paragraph names start in Area A. A paragraph consists of zero or more sentences, each ending with a period. A sentence consists of one or more verbs. Each verb starts in Area B. If the sentence defined by the verbs is written on more than one line, the continued line(s) also start in Area B. A verb is not required to be the first word on a line.

When the program is placed into execution, control begins at the first section or paragraph (following the optional **DECLARATIVES** area) in the program. *Control* means the place at which the program is being executed. Normally, control proceeds sequentially through the program, performing the actions indicated by each verb encountered. Certain verbs, however, may cause control to be altered to some other place in the program. These actions are described in detail in the sections of the manual dealing with COBOL verbs.

6.2 Declaratives

This area starts with

DECLARATIVES.

and ends with

END DECLARATIVES.

Both are written starting in Area A. The area between these statements consists of a number of sections. Each section name is immediately followed by a **USE** statement which specifies an error condition. Should that error condition arise during the execution of the program, then control is passed to the first paragraph following the **USE** statement.

When the execution of the section is complete, control returns to statement following that which caused the error. It should be noted that this mechanism provides a method whereby errors can be trapped, diagnosed, and/or corrective action can be applied. The subsections describing the **USE** statement should be consulted for the specific conditions which may be given in that statement.

Each section in the **DECLARATIVES** area should be considered self-contained for the following reasons:

(1) There can be no reference to a section or paragraph name in the **DECLARATIVES** area from outside that area, except in a **PERFORM** statement.

(2) There can be no reference from within the **DECLARATIVES** area to a section or paragraph name found outside the area.

(3) No action can take place while executing statements in a **DECLARATIVES** section which cause the execution of another **DECLARATIVES** section that had previously been invoked and had not yet returned control to the place of invocation.

6.3 Common Terms

This section contains descriptions of several common terms which will be referenced in the following sections. The explanations are included separately since they apply to a number of statements.

6.3.1 Arithmetic Expressions

Arithmetic expressions are used in various statements in order to specify values which are to be calculated. For example,

```
COMPUTE Z = X * Y + B.
```

is a statement which specifies that the expression

```
X * Y + B
```

be evaluated and that the resultant value is to be assigned to the data item "Z". In the example, the expression is evaluated by multiplying together the values of "X" and "Y" and then adding the value of "B" to produce the final result.

An expression is written as a combination of names of elementary data items, numeric literals, arithmetic operators and parentheses. The rules by which these elements are combined are very similar to the familiar conventions of algebra or arithmetic.

The following *binary operators* (given between two values) may be used in arithmetic expressions

+	add two values
−	subtract second value from first
*	multiply two values
/	divide second value into first
**	raise first value to power of second value

and the following *unary operators* (given in front of a value) may be used:

+	has the effect of multiplying by +1
−	has the effect of multiplying by −1

These operators may be combined with parentheses, names and literals in the manner shown in the following table:

	Name	binary op.	unary op.	(	)
Name		x			x
Binary op.	x		x	x	
Unary op.	x			x	
(	x		x	x	
)		x			x

The table has five columns and rows. An "x" at the intersection of a row and column indicates that an item from the column can immediately follow an item from the row. The "name" item represents both numeric literals and elementary data items.

In addition, the following rules apply:

(1) An expression must start with a name, opening parenthesis or a unary operator.

(2) An expression must end with a name or a closing parenthesis.

(3) The parentheses must be paired such that each closing parenthesis is to the right of the corresponding opening parenthesis.

Operators must be written with a space both before and after the operator.

The *order* in which an expression is evaluated is determined by parentheses in the expression and by the priority of the operators. The following priorities of operators apply:

1 Unary + and −

2 Exponentiation

3 Multiplication and Division

4 Addition and Subtraction

Operations enclosed within parentheses are performed first, with the inner most pairs evaluated before the outer pairs. When parentheses are not used, or parenthesized expressions are at the same level of inclusiveness, the priority of operations determines the order in which the operations are applied.

Consider the following expression, where data items A, B and C have values 4, 6 and 2 respectively:

```
- A ** ( (2 + B) / C ) * 2.5 * C
```

The evaluation proceeds as follows:

```
- A ** ((2 + B) / C) * 2.5 * C

{-4} ** ((2 + B) / C) * 2.5 * C
{-4} ** ( {8}  / C) * 2.5 * C
{-4} ** (    {4}  ) * 2.5 * 2
{256} * 2.5 * 2
{640} * 2
{1280}
```

The result of each operation has been shown in braces ({}).

Arithmetic expressions are calculated with 36 significant digits of internal accuracy. In addition to zero, the absolute values which may be represented range from 10 to the 75-th power to 10 to −75-th power. The exponentiation (raising to a power) operation is an approximation procedure which varies from implementation to implementation; it usually gives results which have fewer (typically 7 or 8) significant digits of precision.

6.3.2 Conditional Expressions

Conditional expressions identify conditions that are tested to enable the object program to select between alternate paths of control depending upon the truth value of the condition. Conditional expressions are specified in the **IF** and **PERFORM** statements. There are two categories of conditions associated with conditional expressions: simple conditions and complex conditions. Each may be enclosed within any number of paired parentheses, in which case its category is not changed.

6.3.2.1 Simple Conditions

The *simple conditions* are the relation, class, condition-name and sign conditions. A simple condition has a truth value of 'true' or 'false'. The inclusion in parentheses of simple conditions does not change the simple truth value.

6.3.2.1.1 Relation Condition

A relation condition causes a comparison of two operands, each of which may be the data item referenced by an identifier, a literal, or the value resulting from an arithmetic expression. A relation condition has a truth value of 'true' if the relation exists between the operands. Comparison of two numeric operands is permitted regardless of the formats specified in their respective **USAGE** clauses. However, for all other comparisons the operands must have the same usage. If either of the operands is a group item, the nonnumeric comparison rules apply.

The general format of a relation condition is as follows:

```
{ identifier              } relation  { identifier              }
{ literal                 }           { literal                 }
{ arithmetic expression   }           { arithmetic expression   }
```

where a "relation" is one of the relational operators:

IS [**NOT**] **GREATER** THAN
IS [**NOT**] **LESS** THAN
IS [**NOT**] **EQUAL** TO
IS [**NOT**] >
IS [**NOT**] <
IS [**NOT**] =

The first operand is called the *subject* of the condition; the second operand is called the *object* of the condition. The relation condition must contain at least one reference to an identifier.

The relational operator specifies the type of comparison to be made in a relation condition. A space must precede and follow each reserved word comprising the relational operator. When used, **NOT** and the next key word or relation character are one relational operator that defines the comparison to be executed for truth value; e.g., **NOT EQUAL** is a truth test for an 'unequal' comparison; **NOT GREATER** is a truth test for an 'equal' or 'less' comparison.

6.3.2.1.1.1 Comparison of Numeric Operands

For operands whose class is numeric, a comparison is made with respect to the algebraic value of the operands. The length of the literal or arithmetic expression operands, in terms of number of digits represented, is not significant. Zero is considered a unique value regardless of the sign.

Comparison of these operands is permitted regardless of the manner in which their usage is described. Unsigned numeric operands are considered positive for purposes of comparison.

6.3.2.1.1.2 Comparison of Nonnumeric Operands

For nonnumeric operands, or one numeric and one nonnumeric operand, a comparison is made with respect to the collating sequence of characters. One of the operands is specified as numeric, it must be an integer data or an integer literal and:

a. If the nonnumeric operand is an elementary data item or a nonnumeric literal, the numeric operand is treated as though it were moved to an elementary alphanumeric data item of the same size as the numeric data item (in terms of standard data format characters), and the contents of this alphanumeric data item were then compared to the nonnumeric operand.

b. If the nonnumeric operand is a group item, the numeric operand is treated as though it were moved to a group of the same size as the numeric data item (in terms of standard data format characters), and the contents of this group item were then compared to the nonnumeric operand.

c. A non-integer numeric operand cannot be compared to a nonnumeric operand.

The size of an operand is the total number of standard data format characters in the operand. Numeric and nonnumeric operands may be compared only when their usage is the same.

There are two cases to consider: operands of equal size and operands of unequal size.

(1) Operands of equal size. If the operands are of equal size, comparison effectively proceeds by comparing characters in corresponding character positions starting from the high order end and continuing until either a pair of unequal characters is encountered or the low order end of the operand is reached, whichever comes first. The operands are determined to be equal if all pairs of characters compare equally through the last pair, when the low order end is reached.

The first encountered pair of unequal characters is compared to determine their relative position in the collating sequence. The operand that contains the character that is positioned higher in the collating sequence is considered to be the greater operand.

(2) Operands of unequal size. If the operands are of unequal size, comparison proceeds as though the shorter operand were extended on the right by sufficient spaces to make the operands of equal size.

6.3.2.1.2 Class Condition

The *class condition* determines whether the operand is numeric, that is, consists entirely of the characters '0', '1', '2', '3', ..., '9', with or without the operational sign, or alphabetic, that is, consists entirely of the characters 'A', 'B', 'C', ..., 'Z', space. The general format for the class condition is as follows:

```
identifier IS { NOT }     { NUMERIC    }
                          { ALPHABETIC }
```

The usage of the operand being tested must be described as display. When used, **NOT** and the next key word specify one class condition that defines the class test to be executed for truth value; e.g. **NOT NUMERIC** is a truth test for determining that an operand is nonnumeric.

The **NUMERIC** test cannot be used with an item whose data description describes the item as alphabetic or as a group item composed of elementary items whose data description indicates the presence of operational sign(s). If the data description of the item being tested does not indicate the presence of an operational sign, the item being tested is determined to be numeric only if the contents are numeric and an operational sign is not present. If the data description of the item does indicate the presence of an operational sign, the item being tested is determined to be numeric only if the contents are numeric and a valid operational sign is present. Valid operational signs for data items described with the **SIGN IS SEPARATE** clause are the standard data format characters, '+' and '−'.

The **ALPHABETIC** test cannot be used with an item whose data description describes the item as numeric. The item being tested is determined to be alphabetic only if the contents consist of any combination of the alphabetic characters 'A' through 'Z' and the space.

6.3.2.1.3 Condition-Name Condition (Conditions Variable)

In a condition-name condition, a conditional variable (see Level 88 data items) is tested to determine whether or not its value is equal to one of the values associated with a condition-name. The general format for the condition-name condition is as follows:

condition-name

If the condition-name is associated with a range or ranges of values, then the conditional variable is tested to determine whether or not its value falls in this range, including the end values.

The rules for comparing a conditional variable with a condition-name value are the same as those specified for relation conditions.

The result of the test is true if one of the values corresponding to the condition-name equals the value of its associated conditional variable.

6.3.2.1.4 Sign Condition

The sign condition determines whether or not the algebraic value of an arithmetic expression is less than, greater than, or equal to zero. The general format for a sign condition is as follows:

```
arithmetic-expression IS { NOT }     { POSITIVE  }
                                     { NEGATIVE  }
                                     { ZERO      }
```

When used, **NOT** and the next key word specify one sign condition that defines the algebraic test to be executed for truth value; e.g., **NOT ZERO** is a truth test for a nonzero (positive or negative) value. An operand is positive if its value is greater than zero, negative if its value is less than zero, and zero if its value is equal to zero. The arithmetic expression must contain at least one reference to a variable.

6.3.2.2 Complex Conditions

A *complex condition* is formed by combining simple conditions, combined conditions and/or complex conditions with logical connectors (logical operators **AND** and **OR**) or negating these conditions with logical negation (the logical operator **NOT**) The truth value of a complex condition, whether parenthesized or not, is that truth value which results from the interaction of all the stated logical operators on the individual truth values of simple conditions, or the intermediate truth values of conditions logically connected or logically negated.

The logical operators and their meanings are:

Logical Operator	*Meaning*
AND	Logical conjunction; the truth value is 'true' if both of the conjoined conditions are true; 'false' if one or both of the conjoined conditions is false.
OR	Logical inclusive OR; the truth value is 'true' if one or both of the included conditions is true; 'false' if both included conditions are false.
NOT	Logical negation or reversal of truth value; the truth value is 'true' if the condition is false; 'false' if the condition is true.

The logical operators must be preceded by a space and followed by a space.

6.3.2.2.1 Negated Simple Conditions

A simple condition is negated through the use of the logical operator **NOT**. The *negated simple condition* effects the opposite truth value for a simple condition. Thus the truth value of a negated simple condition is 'true' if and only if the truth value of the simple condition is 'false'; the truth value of a negated simple condition is 'false' if and only if the truth value of the simple condition is 'true'. The inclusion in parentheses of a negated simple condition does not change the truth value.

The general format for a negated simple condition is:

NOT simple-condition

6.3.2.2.2 Combined and Negated Combined Conditions

A *combined condition* results from connecting conditions with one of the logical operators **AND** or **OR**. The general format of a combined condition is:

```
condition {   { AND   } condition }...
              { OR    }
```

Where 'condition' may be:

(1) A simple condition, or

(2) A negated simple condition, or

(3) A combined condition, or

(4) A negated combined condition; i.e., the **NOT** logical operator followed by a combined condition enclosed within parentheses, or

(5) Combinations of the above, specified according to the rules summarized in the following table. Combinations of Conditions, Logical Operators, and Parentheses.

Although parentheses need never be used when either **AND** or **OR** (but not both) is used exclusively in a combined condition, parentheses may be used to effect a final truth value when a mixture of **AND**, **OR** and **NOT** is used.

The following table indicates the ways in which conditions and logical operators may be combined and parenthesized. There must be a one-to-one correspondence between left and right parentheses such that each left parenthesis is to the left of its corresponding right parenthesis.

Given the following element	Location in conditional expression		In a left-to-right sequence of elements	
	First	Last	Element, when not first, may be immediately preceded by only:	Element, when not last may be immediately followed by only:
condition	Yes	No	OR, NOT, AND, (	OR, AND,)
OR or AND	No	No	condition,)	condition, NOT, (
NOT	Yes	No	OR, AND, (	condition, (
(	Yes	No	OR, NOT, AND, (	condition, NOT, (
)	No	Yes	condition,)	OR, AND,)

Thus, the element pair **OR NOT** is permissible while the pair **NOT OR** is not permissible; "**NOT**" is permissible while **NOT NOT** is not permissible.

6.3.2.2.3 Abbreviated Combined Relation Conditions

When simple or negated simple relation conditions are combined with logical connectives in a consecutive sequence such that a succeeding relation condition contains a subject or subject and relational operator that is common with the preceding relation condition, and no parentheses are used within such a consecutive sequence, any relation condition except the first may be abbreviated by:

(1) The omission of the subject of the relation condition, or

(2) The omission of the subject and relational operator of the relation condition.

The format for an *abbreviated combined relation condition* is:

```
relation-condition {         { AND  } [ NOT ]
                             { OR   }

     [ relational-operator ] object } ...
```

Within a sequence of relation conditions both of the above forms of abbreviation may be used. The effect of using such abbreviations is as if the last preceding stated subject were inserted in place of the omitted subject, and the last stated relational operator were inserted in place of the omitted relational operator. The result of such implied insertion must comply with the rules shown. This insertion of an omitted subject and/or relational operator terminates once a complete simple condition is encountered within a complex condition.

The interpretation applied to the use of the word **NOT** in an abbreviated combined relation condition is as follows:

(1) If the word immediately following **NOT** is **GREATER,** '>', **LESS,** '<', **EQUAL,** '=', then the **NOT** participates as part of the relational operator; otherwise

(2) The **NOT** is interpreted as a logical operator and, therefore, the implied insertion of subject or relational operator results in a negated relation condition.

Some examples of abbreviated combined and negated combined relation conditions and expanded equivalents follow.

Abbreviated Combined Relation Condition	Expanded Equivalent
a > b AND NOT < c OR d	((a > b) AND (a NOT < c)) OR (a NOT < d)
a NOT EQUAL b OR c	(a NOT EQUAL b) OR (a NOT EQUAL c)
NOT a = b OR c	(NOT (a = b)) OR (a = c)
NOT (a GREATER b OR < c)	NOT ((a GREATER b) OR (a < c))
NOT (a NOT > b AND c AND NOT d)	NOT ((((a NOT > b) AND (a NOT > c)) AND (NOT (a NOT > d))))

6.3.2.2.4 Condition Evaluation Rules

Parentheses may be used to specify the order in which individual conditions of complex conditions are to be evaluated when it is necessary to depart from the implied evaluation precedence. Conditions within parentheses are evaluated first, and, within nested parentheses, evaluation proceeds from the least inclusive condition to the most inclusive condition. When parentheses are not used, or parenthesized conditions are at the same level of inclusiveness, the following hierarchical order of logical evaluation is implied until the final truth value is determined:

(1) Values are established for arithmetic expressions.

(2) Truth values for simple conditions are established in the following order:

relation (following the expansion of any abbreviated relation condition)
class
condition-name
switch-status
sign

(3) Truth values for negated simple conditions are established.

(4) Truth values for combined conditions are established: **AND** logical operators, followed by **OR** logical operators.

(5) Truth values for negated combined conditions are established.

(6) When the sequence of evaluation is not completely specified by parentheses, the order of evaluation of consecutive operations of the same hierarchical level is from left to right.

6.4 CORRESPONDING Items

Several COBOL verbs (**ADD**, **SUBTRACT**, **MOVE**) have an optional **CORRESPONDING** or **CORR** keyword which may be used when the verb operands refer to group items. The inclusion of this keyword causes these verbs to act individually upon subordinate items of these group items, when the names of the subordinate items exactly correspond within their group items. Consider the following records:

```
01  A.
    02  B   PIC 99.
    02  D   PIC 99.
    02  E   PIC 99.

01  F.
    02  B   PIC 99.
    02  C   PIC 99.
    02  D   PIC 99.
```

The statement

```
MOVE CORRESPONDING A TO F.
```

is equivalent to the two statements:

```
MOVE            B IN A      TO     B IN F.
MOVE            D IN A      TO     D IN F.
```

Only those fields that have the same names when fully qualified, up to but not including the group items, in the statement are **MOVE**d.

Two items are said to correspond when they are subordinate to the group items named in the statement using **CORRESPONDING** and:

(1) The items do not have **FILLER** as a data name.

(2) The items have the same data name and qualifiers up to the group items named in the statement.

(3) At least one of the items is elementary (**MOVE** statement) or both are elementary (**ADD**, **SUBTRACT** statements).

(4) Neither items contain 66 or 88 level data items.

(5) Neither items have **REDEFINES**, **RENAMES** or **OCCURS** clauses.

The group items in the **CORRESPONDING** statement may have a **REDEFINES** or **OCCURS** clause. Any items, and subordinate items to them, which are subordinate to the group items and have **REDEFINES** or **OCCURS** clauses, are not considered to be corresponding.

6.5 Undefined Values

The Waterloo microCOBOL Interpreter detects as an error any attempt to use an *undefined* value. A data item to which a value has not yet been assigned is said to be undefined.

Other **COBOL** processors may not detect the use of undefined values or may place predictable values into undefined items. It is considered poor programming practice to rely on these nonstandard features.

Chapter 7

Interacting with the Terminal

7.1 Overview

Two COBOL statements may be used to interact with a user of a program, via the terminal. The **DISPLAY** statement may be used to transmit data to be shown upon the terminal screen. The **ACCEPT** statement transfers data entered using the keyboard to a data item.

The various input/output statements may also be used with the terminal screen and keyboard considered to be files. The statements are described in later chapters.

The **ACCEPT** statement may also be used to obtain the current date and time. For completeness, these uses of the **ACCEPT** statement are also described in this chapter.

The next sections describe the **ACCEPT** and **DISPLAY** statements.

7.2 ACCEPT Statement

```
ACCEPT identifier [ FROM   { DATE } ]
                           { TIME }
```

The **ACCEPT** statement may be used to obtain data from the user's terminal or to obtain data representing the current time or the current data. The accepted data is transferred to the data item specified following the **ACCEPT** keyword. This transfer obeys the following rules:

(1) The size of data transferred is the minimum of the accepted data and the size of the accepting data item.

(2) No verification is performed for the appropriateness of the data for the data item in question.

(3) The data is directly transferred. No editing is performed in this transfer.

(4) When the accepted data is shorter than the accepting item, the transfer starts at the leftmost character in the data item. Characters in the data item to which data is not transferred remain unchanged.

Thus, the **ACCEPT** statement may be used to obtain data interactively from the terminal. Caution should be used in this situation, as a portion of the accepting data item will unchanged if less data is transferred than the item can contain.

When the **FROM TIME** clause is specified, the data returned is an eight-character integer value (no sign) representing the number of seconds since midnight. Thus, 2:41 p.m. would be expressed as 14410000.

When the **FROM DATE** clause is specified, the data returned is a six-character integer value. The date of March 9, 1982 would be expressed as 820309. Two digits are used for each of the year of the century, month and day of month.

7.3 DISPLAY Statement

```
DISPLAY   { identifier } [,  { identifier } ] ...
          { literal    }     { literal    }
```

The **DISPLAY** statement may be used to display data upon the terminal. The data to be displayed is given a list following the keyword **DISPLAY**. Each item in the list is either a literal or the name of a data item. In the latter case, the value of the data item is displayed. The data is displayed upon the terminal without any intervening blanks or editing, in the order in which items are given in the list. When the sizes of the items exceeds the size of a line on the terminal, the current line is displayed and the remainder is displayed using another line.

When all or part of a data item has not been assigned a value during the execution of a program, those character positions are said to be *undefined*. Undefined characters are **DISPLAY**ed as question-mark (?) characters.

Chapter 8

MOVE Statement

```
MOVE  { identifier } TO { identifier } [, identifier ] ...
      { literal    }

MOVE  { CORRESPONDING } identifier TO identifier
      { CORR          }
```

The **MOVE** statement is used to transfer data to one or more data areas. When the move involves elementary items, the data may be edited from one representation to another. The contents of this chapter are also important as several other descriptions in the reference manual describe the use of data items as if they had been moved to specific fields in particular ways. A **READ** statement with an **INTO** clause, for example, causes data to be moved from the **FILE SECTION** to a data item. This transfer is accomplished with the same rules as if the data had been **MOVE**d.

When the **CORR** or **CORRESPONDING** keyword is used, corresponding items are moved from the source group item to the target group items. Refer to the section on CORRESPONDING ITEMS for a description of how the corresponding items are selected for a pair of group items. The results of a MOVE with this option are as if the corresponding items had been specified individually in separate **MOVE** statements.

The following rules apply the **MOVE** verb:

(1) The data designated by the literal or identifier following the **MOVE** keyword is moved first to the data items in the order that they follow the

TO keyword. Any *subscripting* or *indexing* associated with an identifier following the **TO** keyword is evaluated immediately before the data is moved to the respective data item.

Any subscripting or indexing associated with the identifier which follows the **MOVE** keyword is evaluated only once, immediately before data is moved to the first of the receiving operands. The result of the statement

```
MOVE a (b) TO b, c (b)
```

is equivalent to:

```
MOVE a (b) TO temp
MOVE temp TO b
MOVE temp TO c (b)
```

where "temp" is an intermediate result item provided by the implementor.

(2) Any **MOVE** in which the sending and receiving items are both elementary items is an *elementary move*. Every elementary item belongs to one of the following categories: numeric, alphabetic, alphanumeric, numeric edited, alphanumeric edited. These categories are described in the section dealing with **PICTURE** strings. Numeric literals belong to the category numeric, and nonnumeric literals belong to the category alphanumeric. The figurative constant **ZERO** belongs to the category numeric. The figurative constant **SPACE** belongs to the category alphabetic. All other figurative constants belong to the category alphanumeric.

The following rules apply to an elementary move between these categories:

a. The figurative constant **SPACE**, a numeric edited, alphanumeric edited, or alphabetic data item must not be moved to a numeric or numeric edited data item.

b. A numeric literal, the figurative constant **ZERO**, a numeric data item or a numeric edited data item must not be moved to an alphabetic data item.

c. A non-integer numeric literal or a non-integer numeric data item must not be moved to an alphanumeric or alphanumeric edited data item.

d. All other elementary moves are legal and are performed according to the rules given in the next rule.

(3) Any necessary *conversion of data* from one form of internal representation to another takes place during legal elementary moves, along with any editing specified for the receiving data item:

a. When an alphanumeric edited or alphanumeric item is a receiving item, alignment and any necessary space filling takes place. If the size of the sending item is greater than the size of the receiving item, the excess characters are truncated on the right after the receiving item is filled. If the sending item is described as being signed numeric, the operational sign will not be moved; if the operational sign occupied a separate character position, the character will not be moved and the size of the sending item will be considered to be one less than its actual size.

b. When a numeric or numeric edited item is the receiving item, alignment by decimal point and any necessary zero-filling takes place, except where zeroes are replaced because of editing requirements.

1) When a signed numeric item is the receiving item, the sign of the sending item is placed in the receiving item. Conversion of the representation of the sign takes place as necessary. If the sending item is unsigned, a positive sign is generated for the receiving item.

2) When an unsigned numeric item is the receiving item, the absolute value of the sending item is moved and no operational sign is generated for the receiving item.

3) When a data item described as alphanumeric is the sending item, data is moved as if the sending item were described as an unsigned numeric integer.

c. When a receiving field is described as alphabetic, justification and any necessary space-filling takes place. If the size of the sending item is greater than the size of the receiving item, the excess characters are truncated on the right after the receiving item is filled.

(4) Any move that is not an elementary move is treated exactly as if it were an alphanumeric to alphanumeric elementary move, except that there is no conversion of data from one form of internal representation to another. In such a move, the receiving area will be filled without consideration for the individual elementary or group items contained within either the sending or receiving area, except as noted in the preceding rule with the **OCCURS** clause.

(5) Data in the following chart summarizes the legality of the various types of **MOVE** statements. The general rule reference indicates the rule that prohibits the move or the behavior of a legal move.

SENDING ITEM	ALPHABETIC	ALPHANUMERIC EDITED ALPHANUMERIC	NUMERIC EDITED NUMERIC
ALPHABETIC	YES/3C	YES/3a	NO/2a
ALPHANUMERIC	YES/3c	YES/3a	YES/3b
ALPHANUMERIC EDITED	YES/3c	YES/3a	NO/3a
NUMERIC INTEGER	NO/2b	YES/3a	YES/3b
NUMERIC NON-INTEGER	NO/2b	NO/2c	YES/3b
NUMERIC EDITED	NO/2b	YES/3a	NO/2a

Chapter 9

Arithmetic Statements

9.1 Overview

This chapter is concerned with the statements that cause *computations* to be performed and the result saved in data items. The **ADD**, **SUBTRACT**, **MULTIPLY** and **DIVIDE** statements perform the operations indicated by their names. The **COMPUTE** statement causes an arithmetic expression to be evaluated and the resultant value to be stored in data items.

In the next section several common clauses found with the arithmetic statements will be described. The subsequent sections will describe the five arithmetic statements.

9.2 Common Terms

In all the arithmetic statements the optional **ROUNDED** keyword and/or the optional **SIZE ERROR** clause may be specified. These features are described in this section to avoid redundant explanations with the description of each verb.

9.2.1 ROUNDED

In the arithmetic statements any data item which is specified to receive a value (except the **REMAINDER** identifier in **DIVIDE**) may be given with the **ROUNDED** keyword immediately following the data name. This keyword causes values to be assigned to these identifiers with the number of decimal places rounded to the number of decimal places in the data item. In the absence of the keyword, the value to be assigned is truncated to the number of decimal places in the receiving data item.

When the low-order integer positions of the receiving data item are represented by the character 'P' in its **PICTURE**, the rounding or truncation occurs relative to the rightmost integer position for which storage is allocated.

9.2.2 SIZE ERROR

ON **SIZE ERROR** imperative-statement

A size error condition exists when the absolute value of a result exceeds the capacity of a data item to contain the value (after decimal point alignment). Division by zero always causes this condition. The size error condition applies to final results only, except for the **MULTIPLY** and **DIVIDE** statements, in which cases the condition applies to intermediate results as well.

When the **SIZE ERROR** clause is not specified for a statement, results causing the error are truncated on the left (after decimal point alignment) for assignment to data items. When the clause is specified, data items for which the condition applies are left unchanged and the imperative sentence specified in the clause is executed.

Receiving data items, for which no size error condition is activated, receive data before the imperative statement in the **SIZE ERROR** clause is executed. Thus, if there are multiple receiving values, either specified or resulting from a **CORRESPONDING** clause, those data items for which there is no size error condition will all receive values. Consequently, if the size error condition was detected for any of the receiving values, then the imperative statement in that clause is executed.

9.2.3 Composite of Operands

The term "composite of operands" is used to describe the computational size of a number of numeric operands. The value is calculated as the size of a hypothetical data item resulting from the super imposition of the operands aligned on this decimal point. For example, consider the following items:

```
77  A PIC 999V99
77  B PIC 9999V9
77  C PIC 9V99999
```

The hypothetical operand would have a picture specification of 9999V99999 and would require 9 digits.

The **ADD, SUBTRACT, MULTIPLY** and **DIVIDE** statements all require that the composite of operands not exceed 18 digits. Refer to the descriptions of these statements for the details of which operands are used in this calculation.

9.2.4 ADD Statement

```
ADD  { identifier } [,  { identifier } ] ...
     { literal    }     { literal    }

   TO identifier [ ROUNDED ] [, identifier [ ROUNDED ] ...

   [;  ON SIZE ERROR imperative statement ]


ADD  { identifier }, { identifier } [,  { identifier } ] ...
     { literal    }  { literal    }     {literal       ]

   GIVING identifier [ ROUNDED ] [ identifier [ ROUNDED ] ]...

   [;  ON SIZE ERROR imperative statement ]


ADD  { CORRESPONDING  } identifier TO identifier [ ROUNDED ]
     { CORR           }

   [;  ON SIZE ERROR imperative statement ]
```

This statement calculates the sum of a number of elementary numeric operands and then stores that sum. The sum is stored in each data item specified by the **TO** phrase or by the **GIVING** phase. The sum is calculated either from all operands (**TO** used) or from the operands preceding the **GIVING** keyword.

When the **CORRESPONDING** or **CORR** keyword is specified, the effect is the same as if all corresponding data items (see **CORRESPONDING** section) were specified in appropriate **ADD** statements. An exception to this effect is the **SIZE ERROR** clause which, if specified, is executed only once at the end of the list of conceptual **ADD** statements, if a size error condition was detected in any of them (see **SIZE ERROR**).

The results assigned to data items may be rounded (see **ROUND**).

The composite of operands (see COMPOSITE OF OPERANDS) is determined for all operands used to produce the sum. When the **CORRESPONDING** or **CORR** keyword is given, the composite is determined for each pair of corresponding items. The composite of operands must not exceed 18 digits.

9.2.5 COMPUTE Statement

COMPUTE identifier [**ROUNDED**] [, identifier [**ROUNDED**]] ...

= arithmetic-expression

[; ON **SIZE ERROR** imperative statement]

The **COMPUTE** statement evaluates an arithmetic expression (see ARITHMETIC EXPRESSIONS) and assigns the resultant value to one or more data items. These receiving items must be elementary numeric or numeric edited items. The value assigned to these items may be rounded (see **ROUNDED**). A size error condition may cause the execution of the imperative statement given in a **SIZE ERROR** clause (see **SIZE ERROR**).

The expression is calculated only once per **COMPUTE** statement. The value is then assigned to each of the data items specified to the left of the assignment operator (=).

9.2.6 DIVIDE Statement

```
DIVIDE  { identifier  } INTO identifier [ ROUNDED ]
        { literal     }

   [, identifier [ ROUNDED ] ] ...

   [; ON SIZE ERROR imperative statement ]


DIVIDE  {identifier   } INTO { identifier  }
        { literal     }      {literal      }

   GIVING identifier [ ROUNDED ] [, identifier [ ROUNDED ] ]...

   [; ON SIZE ERROR imperative statement ]


DIVIDE  { identifier  } BY { identifier  }
        { literal     }    { literal     }

   GIVING identifier [ ROUNDED ] [, identifier [ ROUNDED ] ]...

   [; ON SIZE ERROR imperative statement ]


DIVIDE  { identifier  } INTO { identifier  }
        { literal     }      { literal     }

   GIVING identifier [ ROUNDED ] REMAINDER identifier

   [; ON SIZE ERROR imperative statement ]


DIVIDE  { identifier  } BY { identifier  }
        { literal     }    { literal     }

   GIVING identifier [ ROUNDED ] REMAINDER identifier

   [; ON SIZE ERROR imperative statement ]
```

The **DIVIDE** statement causes an elementary numeric data item or numeric literal to be divided into one or more numeric operands to produce one or more quotients and, optionally, a remainder.

When the **INTO** clause is specified, the operand following the **DIVIDE** keyword is divided into the operand(s) following the **INTO** keyword. When the **BY** keyword is given, the operand following the **BY** keyword is divided into the operand following the **DIVIDE** keyword.

When a list of data items follows the **INTO** keyword, each of the quotients formed replaces the respective data items used in the computations. When the **GIVING** keyword is present, the quotient is assigned to each of the data items following that keyword. If the **REMAINDER** keyword is present, the computed remainder is assigned to the data-item following that keyword.

All data items to receive quotients may be followed by the **ROUNDED** keyword, in which case those items receive rounded values (see **ROUNDED**).

Execution of the statement may cause a size error to be detected when assigning a quotient(s) or remainders. When a size error occurs for the quotient, the data item specified in the optional **REMAINDER** clause is unchanged.

The composite of operands (see COMPOSITE OF OPERANDS) is determined for all data items in the statement which receive a quotient. This value must not exceed 18 digits.

9.2.7 MULTIPLY Statement

```
MULTIPLY  { identifier } BY  { identifier } [ ROUNDED ]
          { literal    }

    [, identifier [ ROUNDED ] ] . . .

    [; ON SIZE ERROR imperative statement ]


MULTIPLY  { identifier } BY  { identifier }
          { literal    }     { literal    }

    GIVING identifier [ ROUNDED ]

         [, identifier [ ROUNDED ] ] . . .

    [; ON SIZE ERROR imperative statement ]
```

The **MULTIPLY** statement causes a number of numeric operands to be multiplied together and the resulting product to be assigned to one or more items. Operands following the **MULTIPLY** and **BY** keywords must be numeric; operands following the **GIVING** keyword must be numeric or numeric edited. All operands must be elementary.

When the **GIVING** clause is specified, the product formed by multiplying the operands following the **MULTIPLY** and **BY** keywords is assigned to each data item in the list following the **GIVING** keyword. When the **GIVING** clause is omitted, the operand following the **MULTIPLY** keyword is multiplied by each data item in the list and each product is assigned to the respective data item in the list.

Each data item receiving a product may receive a rounded value if the data name is followed by the **ROUNDED** keyword (see **ROUNDED**).

The composite of operands (see COMPOSITE OF OPERANDS) is determined using all the receiving data items. This value may not exceed 18 digits.

9.2.8 SUBTRACT Statement

```
SUBTRACT  { identifier } [,  { identifier } ... ]
          { literal    }     { literal    }

     FROM identifier [ ROUNDED ] [ identifier [ ROUNDED ] ]...

     [; ON SIZE ERROR imperative statement ]


SUBTRACT  { identifier } [,  { identifier } ... ]
          { literal    }     {literal     }

     FROM  { identifier }
           { literal    }

     GIVING identifier [ ROUNDED ]

          [, identifier [ ROUNDED ] ] ...

     [; ON SIZE ERROR imperative statement ]


SUBTRACT  { CORRESPONDING  } identifier
          { CORR           }

     FROM identifier [ ROUNDED ]

     [; ON SIZE ERROR imperative statement ]
```

The **SUBTRACT** statement subtracts a value or a number of values from a number of values and stores the result in a data item. When the **GIVING** keyword is present, the sum of the values following the **SUBTRACT** keyword are subtracted from the value following the **FROM** keyword and the result is placed in each of the data items following the **GIVING** keyword. Otherwise, the sum of the values following the **SUBTRACT** keyword is subtracted from each of identifiers following the **FROM** keyword and that difference is stored in each of the respective identifiers.

When the **CORRESPONDING** or **CORR** keyword is specified, the effect is the same as if all corresponding (see **CORRESPONDING**) data items were specified in **SUBTRACT** statements. An exception to this effect is the **SIZE ERROR** clause which, if specified, is executed only at the end of the conceptual **SUBTRACT** statements, provided the size error condition was detected in any of the (see **SIZE ERROR**).

The results assigned to data items may be rounded (see **ROUND**).

The composite of operands (see **COMPOSITE OF OPERANDS**) is determined for all operands except those following the **GIVING** keyword; when **CORRESPONDING** or **CORR** is given, the composite of operands is determined for each corresponding pair. This value cannot exceed 18 digits.

Chapter 10

Sections and Paragraphs

10.1 Overview

When a program begins execution, the first statement to be executed is the first statement in the program following the optional **DECLARATIVES** area. Normally, the next statement to be executed is the one immediately following the one just completed. Several statements, however, may cause control to change to some other place in the programs. These statements, **GO**, **PERFORM** and **EXIT**, are described in this chapter. The **STOP** statement (halts or suspends execution of a program) is also described. The **ALTER** statement is associated with the **GO** statement and so is described. Other statements which cause control to vary from normal sequential execution include the **IF** statement and other conditional statements which are described elsewhere.

10.2 Procedure Names

The **PROCEDURE DIVISION** is organized either as a group of paragraphs or as a group of sections containing paragraphs. Every paragraph, except possibly the first in the **PROCEDURE DIVISION** or a section, has a name (written in Area A) preceding it. Every section also has a name.

The names of sections and paragraphs are jointly called *procedure names*. These names are important because they are referenced in **GO**, **PERFORM** and **ALTER** statements to specify how control is to be altered from the normal sequential execution of statements.

Section names must be unique and must not be keywords. Paragraph names must be either unique or capable of being uniquely qualified using the name of the section in which they are found:

```
paragraph-name  { IN } section-name
                { OF }
```

One of the preceding forms is used to qualify a paragraph name.

It is considered good programming practice to use names which accurately describe the function performed in sections and paragraphs. In this way the program is more understandable by anybody referring to the source program.

10.3 ALTER Statement

ALTER procedure-name **TO** [**PROCEED TO**] procedure-name

[, procedure-name **TO** [**PROCEED TO**] procedure-name] ...

The **ALTER** statement is used to change the procedure to which control may be transferred using a **GO** statement. The **GO** statement must be the only statement in the paragraph(s) immediately referenced before the **TO** keyword(s). This **GO** statement may not contain a **DEPENDING** clause. The execution of the **ALTER** statement causes any subsequent execution of the **GO** statement(s) to transfer control to the procedure (paragraph or section) named following to **TO** keyword in the **ALTER** statement.

WARNING

Many people, including the authors, discourage or prohibit the use of this verb. It is generally felt that its use tends to decrease the clarity of a program. This is because it is often unclear where the target of a **GO** is located, unless the entire source listing is inspected in detail.

10.4 EXIT Statement

```
EXIT
```

The **EXIT** statement is provided in order to define a procedure name of a given point in the program. It must be the only sentence in a paragraph. The statement has no effect while the program executes.

The statement is often used in a paragraph which is the second paragraph of a **PERFORM-THROUGH** verb (see **PERFORM**). In this way, paragraphs may be added to or deleted from the group of performed paragraphs.

10.5 GO Statement

```
GO TO [ procedure-name ]


GO TO procedure-name [, procedure-name ] ...

    DEPENDING ON identifier
```

The **GO** or **GOTO** statement may be used to transfer control to another part of the **PROCEDURE DIVISION.** When the form

GO TO procedure-name

is used (and an **ALTER** does not apply for the statement), the execution of the statement causes control to be transferred to the point in the program indicated by the procedure name.

No procedure names are given with the **GO** statement, only when the **GO** statement is to be used in conjunction with **ALTER** statements (see **ALTER**). The **DEPENDING** clause is used when one of a number of procedures is to be selected. In this situation, the value of the identifier following that keyword is used to determine to which procedures control is transferred. When the value of this identifier is 1 the first procedure in the list receives control; when it is 2 the second procedure receives control; and so forth. When the value is not an unsigned positive integer value or when the value exceeds the number of procedures in the list, control passes to the next statement according to the normal sequential execution of *statements.*

10.6 PERFORM Statement

```
PERFORM procedure [   { THROUGH  } procedure ]
                      { THRU     }

      [   { identifier  } TIMES ]
          { number      }


PERFORM procedure [   { THROUGH  } procedure ]
                      { THRU     }

      [   UNTIL condition ]


PERFORM procedure [   { THROUGH  } procedure ]
                      { THRU     }

      VARYING  { identifier  } FROM { identifier   }
               { literal     }      { index-name   }
                                    { literal      }

      BY  { identifier  } UNTIL condition
          { literal     }


      [ AFTER  { identifier   } FROM { identifier   }
               { index-name   }      { index-name   }
                                     { literal      }

      BY  { identifier  } UNTIL condition
          { literal     }
[ AFTER  { identifier   } FROM { identifier   }
         { index-name }        { index-name }
                               { literal      }

  BY  { identifier } UNTIL condition ] ]
      { literal    }
```

The **PERFORM** statement is used to transfer control to one or more procedures. The statement differs from the **GO** statement in that control implicitly returns to the point of **PERFORM** when the execution of the **PERFORMed** procedures is complete.

The simplest forms of the **PERFORM** verb are as follows:

```
PERFORM paragraph
PERFORM section
```

The **PERFORM** verb causes control to pass to the paragraph or section referenced. When the last statement in that paragraph or section has been executed, control passes to the statement following the initial **PERFORM** statement.

When the **THROUGH** or **THRU** clause is used

```
PERFORM procedure THRU procedure
```

the procedure following the **PERFORM** keyword is passed control. Control returns to the statement following the **PERFORM** verb when the procedure following the **THRU** or **THROUGH** keyword has been executed.

When the **TIMES** keyword is present, the procedures given in the statement are executed the number of times indicated by the literal or the value of the elementary numeric integer data item preceding the **TIMES** keyword. When the value is a positive integer, the procedures are executed that number of times and then control continues to the next statement in the normal sequential manner of execution. When the value is non-positive, no procedures are **PERFORMed** by the statement.

The **UNTIL** clause (without any **VARYING** clauses) causes the indicated procedures to be **PERFORMed** until the associated condition becomes true. The statement

```
PERFORM P1 [THRU P2]
      UNTIL condition
```

is equivalent to the following group of statements:

```
L1.
     IF condition GO TO L2.
     PERFORM P1 [THRU P2].
     GO TO L1.
L2.
```

It should be noted that the **UNTIL** condition is tested before each **PERFORM** takes place. Thus, if the condition is initially true, no procedures would be **PERFORM**ed by the statement.

The **VARYING** phrase is used in conjunction with the **UNTIL** clause to give a data item a sequence of values, one each time the indicated procedures are **PERFORM**ed

```
PERFORM P1 [THRU P2]
    VARYING D FROM V1 BY V2
        UNTIL condition
```

A statement of the preceding form is equivalent to the following pseudo statements:

```
     set D to V1 value.
L1.
     IF condition GO TO L2.
     PERFORM P1 [THRU P2].
     augment D with V2 value.
     GO TO L1.
L2.
```

One or two AFTER clauses may be given. A statement of the form

```
PERFORM P1 [THRU P2]
    VARYING D1 FROM V11 TO V12
        UNTIL condition-1
    AFTER D2 FROM V21 TO V22
        UNTIL condition-2
    AFTER D3 FROM V31 TO V32
        UNTIL condition-3
```

is equivalent to the following pseudo statements.

```
        set D1 to V11 value.
        set D2 to V21 value.
        set D3 to V31 value.
    L1.
        IF condition-1 GO TO L6.
    L2.
        IF condition-2 GO TO L5.
    L3.
        IF condition-3 GO TO L4.
        PERFORM P1 [THRU P2].
        augment D3 with V32 value.
        GO TO L3.
    L4.
        set D3 to V31 value.
        augment D2 with V22 value.
        GO TO L2.
    L5.
        set D2 to V21 value.
        augment D1 with V12 value.
        GO TO L1.
    L6.
```

When an index name occurs in a **VARYING**, **AFTER** or **FROM** phase, values are placed in the data item in the associated **VARYING** or **AFTER** phase according to the rules of the **SET** statement (see **SET**). Otherwise, data items are initialized according to the rules of the **MOVE** statement (see **MOVE**) and augmented according to the rules of the **ADD** statement (see **ADD**).

A **PERFORM**ed procedure or group of **PERFORM**ed procedures may themselves contain **PERFORM** statements. These statements must **PERFORM** procedures that are completely excluded from procedures initially being actively **PERFORM**ed or, in the case of **THRU** or **THROUGH**, must reference procedures that are all actively being **PERFORM**ed, excluding the actual procedures referenced in the first **PERFORM-THROUGH** statement.

10.7 STOP Statement

```
STOP  { RUN     }
      { literal }
```

The **STOP** statement is used to halt execution of a COBOL program, completely or temporarily. When the **RUN** keyword follows the verb, the execution of the program is completed. When a literal follows the verb, the literal is displayed and the Debugger is entered (see DEBUGGER).

Chapter 11

IF Statement

```
IF condition;   { statement          } [; ELSE  { statement
                { NEXT SENTENCE  }               { NEXT SENTENCE
```

11.1 Overview

The execution of an **IF** statement causes the condition following the **IF** keyword to be evaluated. When the condition is true, the statement following the condition is executed; when the condition is false and the **ELSE** keyword is present, the statement following the **ELSE** is executed. Control normally continues following the **IF** statement, regardless of whether the condition was true or false.

The syntactic definition of the **IF** statement specifies that a statement may be given following either the condition or the **ELSE** keyword. A statement may involve several verbs. Thus,

```
MOVE A1 to B
ADD 2 to C
DISPLAY Q
```

is a statement involving three verbs. It is common to use a statement of this nature with an **IF**. It should be noted that the statement *does not contain* a period (.) character. Because this statement may contain several more elementary statements, the statement following the condition is be called the *true range* of the **IF**. The statement following the **ELSE** keyword is called the *false range* of the **IF**.

The **NEXT SENTENCE** clause may be specified in place of either the statement following the condition or following the **ELSE** keyword. The execution of this clause has no effect. It is useful, however, in permitting the full form of the **IF** statement to be coded. This is often necessary in nested **IF**'s, to be discussed later.

The examples in this chapter will make use of indentation. This is an important convention used when coding programs. Indentation is designed to increase the clarity of the program by making obvious the structure of it. It is not required by the COBOL language nor does it convey any special information to the COBOL processor. The examples could be written, in a less understandable format, without any indentation.

11.2 Simple IF

IF condition; statement

This simplest form of the **IF** statement has no **ELSE** clause. When the condition is evaluated as true, the true range following the condition is executed and then control continues following the **IF** statement; when the condition is false, the true range following the condition is not executed and control continues following the **IF** statement. Thus,

```
if colour = 14
     display 'peach'.
```

causes the **DISPLAY** statement to be executed only when the value of "colour" is 14. The true range of the **IF** is that single statement.

The end of the **IF** statement and of the true range is determined by the period (.) character. Several statements may be found in the true range:

```
if colour = 14
     add 1 to peach-count
     display "peach".
```

In the example, two statements will be executed if "colour" has a value of 14. Note that the period (.) character is given only after the true range.

In order to increase the clarity of the program, it is a common practice to indent the statements that are to be conditionally executed. Various indentation conventions can be used; what is important is that a consistent method be used throughout an entire program. In this way, it is obvious which statements are to be executed when the condition is true. It becomes easy to visually verify that a period follows only the last statement in the sequence.

11.3 ELSE Clause

```
IF condition;   { statement        } [; ELSE  { statement
                { NEXT SENTENCE  }           { NEXT SENTENCE
```

The more general form of an **IF** statement involves an **ELSE** clause. In this case, the execution of the **IF** causes one of two ranges of statements to be selected for execution. When the condition is true, the range following the condition is executed; otherwise, the range following the **ELSE** is executed. Following the execution of one of these ranges, control normally continues following the false range. The **IF** statement

```
IF SALARY > 50000.00
    DISPLAY 'Executive'
ELSE
    DISPLAY 'Worker'.
```

causes "Executive" to be **DISPLAY**ed when the value of "SALARY" exceeds 50,000; otherwise, "Worker" is **DISPLAY**ed.

Several statements may form the true or false ranges:

```
IF SALARY > 50000.00
    ADD 1 TO EXEC-COUNT
    DISPLAY 'Executive'
ELSE
    ADD 1 TO WORKER-COUNT
    DISPLAY "Worker".
```

Again, it is considered good style to indent both groups of statements. A period is given only after the false range.

11.4 Nested IF

An **IF** statement may itself be one of statements in the the true or false range of an **IF.** In this case, the **IF** is said to be *nested*:

```
if salary > 50000.00
     add 1 to high-priced
     if job = 'VP'
          add 1 to vp-count
     else
          add 1 to exec-count
else
     add 1 to worker-count.
```

In the example, an **IF** is nested inside the true range of the outer **IF.** The end of the nested **IF** statement is determined by the **ELSE** of the outer **IF** statement (the second **ELSE** in the example).

An **IF** statement may also be nested inside the false range of an **IF**:

```
if salary > 50000.00
     add 1 to executive-count
else
     if job = 'clerk'
          add 1 to clerk-count
     else
          add 1 to worker-count.
```

In the example, the period (.) character determines the end of both the inner and the outer **IF**'s.

It becomes particularly important to use a consistent style of indentation with nested **IF**'s. In this way, the structure of the program is clearly indicated.

The COBOL language permits an **IF** to be nested only at end of the true or false range of an **IF**. This is because of the way the end of an **IF** statement is determined. The end is determined by encountering a period (.) character or by encountering an **ELSE** keyword for an enclosing **IF** statement. Consider the following:

```
if salary > 50000
    if job = "VP"
        display 'VICE-PRESIDENT'
    else
        display 'EXECUTIVE'
    display salary
else
    display 'WORKER'.
```

The indentation indicates that "VICE-PRESIDENT" and then the value of "salary" should be **DISPLAY**ed when the values of "salary" and "job" are 51000 and "VP" respectively. However, only "VICE-PRESIDENT" would be **DISPLAY**ed. This is because the statement

```
display salary
```

is part of the false range of the nested **IF**. The desired effect may be obtained by placing the inner **IF** in a separate paragraph to be **PERFORM**ed from the place it was originally nested:

```
    if salary > 50000
        perform print-job
        display salary
    else
        display 'worker'.
    . . .
print-job.
    if job = "VP"
        display "VICE-PRESIDENT"
    else
        display "EXECUTIVE".
```

This example accomplishes the effect indicated by the indentation of the original example.

One use of the **NEXT SENTENCE** clause is illustrated by the following (erroneous) example:

```
if salary > 50000
    if job = "VP"
        add 1 to vp-count
else
    add 1 to worker-count.
```

The indentation indicates that 1 is to be added to "worker-count" whenever the value of "salary" does not exceed 50000. However, the **ELSE** clause is part of the nested **IF**, not the outer **IF**. This situation may be remedied as follows:

```
if salary > 50000
    if job = "VP"
        add 1 to vp-count
    else
        next sentence
else
    add 1 to worker-count.
```

The **NEXT SENTENCE** clause is used to give the nested **IF** an **ELSE** clause and so the original **ELSE** clause now applies to the outer **ELSE**.

11.5 Multiple Choice

An **IF** with an **ELSE** can be used to select one of two alternatives. By nesting **IF** statements, a choice may be made to select one from many alternatives:

```
if job = 'VP'
    display 'VICE-PRESIDENT'
else
    if job = 'CL'
        display 'CLERK'
    else
        if job = 'SC'
            display 'SECRETARY'
        else
            display 'WORKER'.
```

In the example, one of four messages is **DISPLAY**ed, depending upon the value of "job".

An alternate method of indentation is often used to emphasize the structure of those multiple-choice situations:

```
if job = 'VP'
    display 'VICE-PRESIDENT'
elseif job = 'CL'
    display 'CLERK'
else if job = 'SC'
    display 'SECRETARY'
else
    display 'WORKER'.
```

In this case, the four choices are shown at the same level of indentation. This emphasizes that one of the four is to be selected.

Chapter 12

Sequential Files

12.1 Introduction to Files

Various input/output statements are used to control the *transmission* of data to and from an executing COBOL program. Data within an executing program is kept in data items and manipulated using various COBOL statements such as **MOVE** or **ADD**.

Data outside of programs is organized into *files*. Each file has a *system name* by which it is catalogued in the computer system in which it resides. A file consists of a number of *records*, each one of which is organized into a number of elementary data items. **WRITE** statements are used transmit new records to a file. **REWRITE** statements are used to transmit records to replace existing records in a file. **READ** statements are used to transmit the data in records to the executing COBOL program.

When a file is to accessed, it must first be connected to the program using an **OPEN** statement. A **CLOSE** statement is used to undo this connection when the program has completed accessing the file.

A COBOL identifier, called a *filename*, is used to identify a given file in a program. It should be noted that the filename is not the system filename; the filename is associated with a particular file by using the **SELECT** statement in the **ENVIRONMENT** division or by using the **VALUE** clause of an **FD** in the **DATA** division.

The records in a file may be accessed either sequentially or randomly. By *sequentially* is meant that the records can only be read in the order that they were originally written to file and that the records can only be written in the order in which they are to be stored in the file. By *randomly* is meant that records can be read or written in any order; the number of the specific record to be accessed is established by the value of the data item given in the **RELATIVE KEY** phrase in a **SELECT** statement for the file in question.

Every file to be accessed in a COBOL program must have at least the following components:

(1) A **SELECT** statement in the **ENVIRONMENT** division.

(2) An **FD** in the **DATA DIVISION**.

(3) An **OPEN** statement which is executed prior to any other statements which are executed and cause access to the file.

(4) A **CLOSE** statement which is executed after all accesses to the file have been completed.

Because of the precise rules concerning the accessibility of data in the **FILE SECTION,** many programmers do not directly access the data in the **FILE SECTION.** Instead, records are copied into **WORKING-STORAGE** as they are read (using the **INTO** clause of the **READ**) and copied from **WORKING-STORAGE** as they are written (using the **FROM** clause of the **WRITE** and **REWRITE** statements).

The **FILE SECTION** defines the memory to contain records read from or transmitted to files. The record descriptions in each **FD** describe the memory. Since more than one type of record can be read or written, a number of record descriptions may be given in an **FD.** Unlike data in the **WORKING-STORAGE SECTION** , the data specified by these record descriptions is not always accessible. It is not accessible, for example, before the file has been **OPEN**ed. A **READ** statement makes accessible the record which is read. A **READ** or **REWRITE** statement transmits the record which is currently accessible and then that record becomes unaccessible.

12.2 ENVIRONMENT DIVISION

The **ENVIRONMENT DIVISION,** in general, is explained in full detail in the chapter by that name. In this section, only the part of the division applying to sequential files is discussed. This pertains to the **SELECT** statement in the **FILE-CONTROL** paragraph of the **INPUT-OUTPUT** section. The syntax permitted for the **SELECT** statement is as follows:

```
SELECT [ OPTIONAL ] file-name

    ASSIGN TO literal

    [; ORGANIZATION IS SEQUENTIAL ]

    [; ACCESS MODE IS SEQUENTIAL ]

    [; FILE STATUS IS name ].
```

This restricted form of the **SELECT** statement applies to sequential files only. The meaning of the various phrases is described in the chapter about the **ENVIRONMENT DIVISION** (see **SELECT**).

12.3 DATA DIVISION

The **DATA DIVISION** is completely explained in the chapter by that name. The description of **FD**'s in that chapter completely applies to sequential files. Consequently, no other details are provided in this section.

12.4 PROCEDURE DIVISION

12.4.1 CLOSE Statement

```
CLOSE file-name [  { { REEL } [  { WITH NO REWIND   } ]  } ]
                   { { UNIT }    { FOR REMOVAL      }     }
                   {                                      }
                   { WITH   { NO REWIND }                 }
                   {        { LOCK      }                 }

    [ , file-name ... ] ...
```

The **CLOSE** statement is used to disconnect a file from a program. It should be executed when the program has completed execution of all statements which access that file. Waterloo microCOBOL checks only this syntax for all options shown in the syntactic description at the start of this section. No actions are performed as a result of these options, since many systems do not have facilities to support them.

12.4.2 OPEN Statement

```
OPEN { INPUT  } file-name, [, file-name ] ...
     { OUTPUT }
     { I-O    }
     { EXTEND }

   [ { INPUT  } file-name, [, file-name ] ... ] ...
     { OUTPUT }
     { I-O    }
     { EXTEND }
```

The **OPEN** statement is used to connect a file to the COBOL program. It prepares the file to be accessed using COBOL statements such as **READ** or **WRITE**. It must be executed before the file can be accessed.

Associated with each file name is one of the following **OPEN** modes:

INPUT	The file will only be accessed using the **READ** statement.
OUTPUT	The file will only be accessed with **WRITE** statements. A new file will be created to consist of the records transmitted using **WRITE** statements. The order of records in this new file is the order in which they were written. If the file already exists, the file written will replace the old one.
I-O	The file may only be accessed using **READ**, or **REWRITE** statements.
EXTEND	The file may only be accessed with **WRITE** statements. The records written are added to the end of the file, in the order written.

A file must exist in order to be **OPEN***ed for* **INPUT**, **I-O** *or* **EXTEND**. The file may exist when **OPEN**ed for output; the new file created will replace the existing file.

Once a file has been **CLOSE**d, it may be **OPEN**ed again. Thus, it is possible to **OPEN** a file for **OUTPUT**, create it using **WRITE** statements, **CLOSE** the file, to **OPEN** it for **INPUT**, **READ** the file, and then the **CLOSE** it again.

12.4.3 READ Statement

```
READ file-name RECORD [ INTO identifier ]

    [; AT END imperative statement ]
```

The **READ** statement causes the *next* logical record to be made available for processing by the program. *Made available* means the next record in the file is now available to be accessed in the record description(s) supplied with the **FD** in the **DATA DIVISION** for the file. This data is available until the next **READ** statement (or a **CLOSE** statement) is executed for the file. There is no record made available following the **OPEN** statement for the file or following a **READ** statement which attempts to read past the end of the file.

When the **INTO** clause is present, the data in the record description for the **FD** is transferred to the record in the **INTO** clause, according to the rules of the **MOVE** statement. Any subscripting or indexing associated with the identifier is performed after the record has been read and before it is transferred to the data item.

An **AT END** condition occurs when an attempt is made to read past the end of the file. The following actions occur in the specified order:

(1) If a **FILE STATUS** data item has been specified by the **SELECT** clause in the **ENVIRONMENT DIVISION**, then that data item is given the appropriate value.

(2) If an **AT END** phrase is specified in the **READ** statement, control is transferred to the imperative statement with that phrase. Any **USE** procedure specified for this file is not executed.

(3) If no **AT END** phrase is specified and a **USE** procedure has been specified, then that procedure is executed.

(4) If neither an **AT END** phrase nor a **USE** procedure exist for the file, an error message will be displayed and the program execution is terminated.

It is an error to attempt to read past the end of a file more than once in a program, without a **CLOSE** followed by an **OPEN** for that file.

12.4.4 REWRITE Statement

```
REWRITE record-name [ FROM identifier ]
```

The **REWRITE** statement is used to *replace* a record in an existing file. The record to be replaced is the one made available by the previous successfully-executed **READ** statement for the file. No other intervening input/output operation is permitted for the file in question. The file must have been **OPEN**ed for **I-O**.

When the **FROM** phrase is present, the execution of the **REWRITE** statement is equivalent to:

```
MOVE identifier TO record-name.
REWRITE record-name.
```

Both the record-name and the identifier must not refer to the same storage area.

The logical record made available by a **READ** statement is no longer available to the program once a **REWRITE** statement is executed for that record.

12.4.5 USE Statement

```
USE AFTER STANDARD   { EXCEPTION  } PROCEDURE
                     { ERROR      }

    ON { file-name [, file-name ] ...   }
       { INPUT                          }
       { OUTPUT                         }
       { I-O                            }
       { EXTEND                         }
```

The **USE** statement specifies procedures to be used when input/output errors occur. The **USE** statement must appear in the declaratives section of the **PROCEDURE DIVISION**. It must immediately follow a section header and is followed by a period. The remainder of the section consists of zero or more procedural paragraphs to be executed when the indicated input/output error occurs. The **USE** statement is never executed; it merely defines the conditions calling for the execution of **USE** procedures. The keywords **ERROR** and **EXCEPTION** are synonymous and may be used interchangeably.

The **USE** procedure is invoked for the input/output errors specified following the optional **ON** keyword:

FILENAME	All input/output errors for the file.
INPUT	All input/output errors for files **OPEN**ed for **INPUT**.
OUTPUT	All input/output errors for files **OPEN**ed for **OUTPUT**.
I-O	All input/output errors for files **OPEN**ed for **I-O**.
EXTEND	All input/output errors for files **OPEN**ed for **EXTEND**.

The **USE** procedure is not executed when the input/output statement causing the error contains a clause, such as **AT END**, to handle the condition.

For additional rules concerning declaratives, see the section entitled **DECLARATIVES**.

12.4.6 WRITE Statement

```
    WRITE record-name [ FROM identifier ]

      [  {BEFORE } ADVANCING { { identifier  } [  { LINES } ]  }]
         { AFTER }           { { number      }    { LINE  }    }
                             {                                 }
                             { PAGE                            }
```

The **WRITE** statement is used to release a record to an **OUTPUT** or **EXTEND** file. *To release means* that the record is conceptually transmitted to the file. The record-name is the name of a logical record used in the **FD** for the file in question.

Immediately after a file is successfully **OPEN**ed for **OUTPUT**, the logical record(s) in the **FD** is available to receive data. A **MOVE** statement, for example, may be used to move data to the logical record. A **WRITE** statement causes the record in the **FD** to be conceptually transmitted to the file. The data in that record is no longer available. Any data subsequently moved to a logical record(s) in the **FD** will be used to compose the next logical record, if any.

When the **FROM** phrase is present, the statement is equivalent to the following statements:

```
MOVE identifier TO record-name
WRITE record-name
```

The **WRITE** statement in the preceding example is to be understood to contain any **ADVANCING** phrase that occurred in the original **WRITE** statement.

The use of the **ADVANCING** phrase is used to control the vertical spacing of records in an output file. It must be specified in all **WRITE** statements for a file or in none of them. When the **BEFORE** keyword is used, the record is written and then the positioning occurs; when the **AFTER** keyword is used, the positioning occurs and then the record is written. The positioning can be to the top of a page (**PAGE** keyword given) or by a number of lines. The first character in every record written with the **ADVANCING** option is reserved for use in vertical positioning, often called *carriage control*.

The positioning indicated by the **ADVANCING** phrase is accomplished in system-dependent manners (see SYSTEM DEPENDENCIES). Generally speaking, there are two different situations:

(1) *Terminal*: The system will attempt to clear the screen when **ADVANCING PAGE** is used and will write blank lines for other positioning. The carriage-control character, at the start of the record, is not displayed upon the screen.

(2) *Carriage-control Files*: These files will have an extra character appended to the front of some or all records. The extra (carriage-control) character is used by the computer hardware to provide vertical positioning on printed pages.

When the **ADVANCING** clause is not specified for a file, the first character in each record is normally transmitted unchanged to the file in question. An exception to this rule occurs when microCOBOL can recognize that a file, such as a printer, will require the first character for positioning. In this case, the following control characters in the leftmost position of a record have the indicated meaning:

'1'	ADVANCE PAGE
'+'	ADVANCE ZERO LINES (overprint)
' '	ADVANCE 1 LINE
'0'	ADVANCE 2 LINES
'-'	ADVANCE 3 LINES

Any character not specified in the preceding is treated as a space character. The detection of these special files is system dependent (see SYSTEM DEPENDENCIES for a description of the files detected in each system).

Chapter 13

Relative Files

13.1 Overview

The concept of files, in general, is introduced in the first section of the chapter about sequential files. The records in a sequential file may be only read or written consecutively. Relative files provide the capability to access records in any order.

When records are to be accessed in *random* or non-sequential order, the position of a record to be accessed is taken from a special data item. The **RELATIVE KEY** phrase of a **SELECT** statement specifies the data item which contains the current record pointer for the file. This is a positive integer value specifying the number of the record to be read or written. The records in the file are numbered consecutively with the initial record at position one. Thus, to read or write a specific record the **RELATIVE KEY** data item should be assigned a number indicating the position of the record to be accessed.

Relative files can also be accessed *sequentially* in much the same way as is discussed in the chapter about Sequential Files. The **ACCESS** clause of the **SELECT** statement specifies exactly how the file is to accessed. These files may be accessed sequentially, relatively, or in combination called dynamic access. When records are accessed sequentially (sequential or dynamic access), special forms of the **READ** and **WRITE** statements are used to indicate that the **RELATIVE KEY** data item is not required.

13.2 ENVIRONMENT DIVISION

The **ENVIRONMENT DIVISION**, in general, is explained in full detail in the chapter by that name. In this section, only the part of the division applying to relative files is discussed. This pertains to the **SELECT** statement in the **FILE-CONTROL** paragraph of the **INPUT-OUTPUT** section. The syntax permitted for the **SELECT** statement is as follows:

```
SELECT [ OPTIONAL ] file-name

    ASSIGN TO literal

    [; ORGANIZATION IS { RELATIVE } ]

    [; ACCESS MODE IS  { SEQUENTIAL [,RELATIVE KEY IS name ]  }]
                       {                                      }
                       { {RANDOM  } ,RELATIVE KEY IS name     }
                       { {DYNAMIC }                           }

    [; FILE STATUS IS name ].
```

This restricted form of the **SELECT** statement applies only to relative files. The meaning of the various phrases is described in the chapter about the **ENVIRONMENT DIVISION** (see **SELECT**).

13.3 DATA DIVISION

The **DATA DIVISION** is completely explained in the chapter by that name. The description of **FD**'s in that chapter completely applies to relative files. Consequently, no other details are provided in this section.

13.4 PROCEDURE DIVISION

13.4.1 CLOSE Statement

```
CLOSE filename [ WITH LOCK ] [, filename [ WITH LOCK ] ] ...
```

The ***CLOSE*** *statement is used to disconnect a file from a program. It should be* executed when the program has completed execution of all statements which access that file. Waterloo microCOBOL checks only this syntax for all options shown in the syntactic description at the start of this section. No actions are performed as a result of these options, since many systems do not have facilities to support them.

13.4.2 OPEN Statement

```
OPEN { INPUT  } file-name, [, file-name ] ...
     { OUTPUT }
     { I-O    }

   [ { INPUT  } file-name, [, file-name ] ... ] ...
     { OUTPUT }
     { I-O    }
```

The **OPEN** statement is used to connect a file to the COBOL program. It prepares the file to the accessed using COBOL statements such as **READ** or **WRITE**. It must be executed for a file before the file can be accessed.

Associated with each file name is one of the following **OPEN** modes:

INPUT	The file will only be accessed using the **READ** statement.
OUTPUT	The file will only be accessed with **WRITE** statements. A new file will be created to consist of the records transmitted using **WRITE** statements. The order of records in this new file is the order in which they were written. If the file already exists, the file written will replace the old one.
I-O	The file may only be accessed using **READ**, or **REWRITE** statements.

A file must exist in order to be **OPEN**ed for **INPUT** or **I-O**. The file may exist when **OPEN**ed for output; the new file created will replace the existing file.

Once a file has been **CLOSE**d, it may be **OPEN**ed again. Thus, it is possible to **OPEN** a file for **OUTPUT**, create it using **WRITE** statements, **CLOSE** the file, to **OPEN** it for **INPUT**, **READ** the file, and then the **CLOSE** it again.

13.4.3 READ Statement

READ file-name [**NEXT** RECORD] [**INTO** identifier]

[; AT **END** imperative statement]

READ file-name RECORD [**INTO** identifier]

[; **INVALID** KEY imperative statement]

The **READ** statement causes the *next* logical record to be made available for processing by the program. By *made available* means that the next record in the file is now available to be accessed in the record description(s) supplied with the **FD** in the **DATA DIVISION** for the file. This data is available until the next **READ** statement (or a **CLOSE** statement) is executed for the file. There is no record made available following the **OPEN** statement for the file or following a **READ** statement which attempts to read past the end of the file.

When the **INTO** clause is present, the data in the record description for the **FD** is transferred to the record in the **INTO** clause, according to the rules of the **MOVE** statement. Any subscripting or indexing associated with the identifier is performed after the record has been read and before it is moved to the data item.

When the **ACCESS** is **SEQUENTIAL**, the records are retrieved sequentially in order that they are located in the file. The **INVALID KEY** clause may not be specified.

An **AT END** condition occurs when an attempt is made to read past the end of the file. The following actions occur in the specified order:

(1) If a **FILE STATUS** data item has been specified by the **SELECT** clause in the **ENVIRONMENT DIVISION**, then that data item is given the appropriate value.

(2) If an **AT END** phrase is specified in the **READ** statement, control is transferred to the imperative statement with that phrase. Any **USE** procedure specified for this file is not executed.

(3) If no **AT END** phrase is specified and a **USE** procedure has been specified, then that procedure is executed.

(4) If neither an **AT END** phrase nor a **USE** procedure exist for the file, an error message will be displayed and the program execution is terminated.

It is an error to attempt to read past the end of a file more than once in a program, without a **CLOSE** followed by an **OPEN** for that file.

When the **ACCESS** is **RELATIVE** neither the **NEXT** keyword nor the **AT END** clause may be specified. The record to be read is located at the position indicated by the data item named **RELATIVE KEY** clause in the **SELECT** statement for the file. If the file does not contain a record at the indicated position, then the **INVALID KEY** condition is detected. The following actions occur in the indicated order:

(1) A value is placed into the **FILE STATUS** data item, if specified, for the file, to indicate the **INVALID KEY** condition.

(2) If an **INVALID KEY** clause is specified for the statement, control is transferred to the imperative statement specified in this clause. Any **USE** procedure specified for this file is not executed.

(3) If an **INVALID KEY** clause is not specified and an appropriate **USE** statement exists for the file, then the indicated **USE** procedure is executed.

(4) If neither an **INVALID KEY** clause nor an appropriate **USE** statement exist, an error message is displayed and the program execution is terminated.

When the **ACCESS** is **DYNAMIC**, the file may be read sequentially or relatively. A sequential **READ** statement specifies the **NEXT** keyword and optionally the **AT END** clause. This statement behaves in the manner of a **READ** statement for **SEQUENTIAL ACCESS** described above. A relative **READ** may not specify the **NEXT** keyword nor the **AT END** clause, but may specify a **INVALID KEY** clause. This statement behaves in the same way as the relative **READ** statement described above.

13.4.4 REWRITE Statement

REWRITE record-name [**FROM** identifier]

[; **INVALID** KEY imperative statement]

The **REWRITE** statement is used to *replace* a record in an existing file. When the **ACCESS** is **SEQUENTIAL**, the record to be replaced is the one made available by the previous successfully executed **READ** statement for the file. No other intervening input/output operation may have been executed for the file in question. The file must have been **OPEN**ed for **I-O.** When the **ACCESS** is either **RELATIVE** or **DYNAMIC**, the record to be replaced is determined by the value of the data item given in the **RELATIVE KEY** clause for the file. When the file does not contain such a record to be updated, the **INVALID KEY** condition is detected and processed (the action to be performed is described in the section about the **READ** statement). The file must be **OPEN**ed for **I-O**.

When the **FROM** phrase is present, the execution of the **REWRITE** statement is equivalent to:

```
MOVE identifier TO record-name.
REWRITE record-name.
```

Both the record-name and the identifier must not refer to the same storage area.

The logical record made available by a **READ** statement is no longer available to the program once a **REWRITE** statement is executed for that record.

13.4.5 USE Statement

```
USE AFTER STANDARD { EXCEPTION } PROCEDURE
                   { ERROR     }

    ON{ file-name [, file-name ] ...  }
      { INPUT                         }
      { OUTPUT                        }
      { I-O                           }
```

The **USE** statement specifies procedures to be used when input/output errors arise. A **USE** statement must appear in the declaratives section of the **PROCEDURE DIVISION**. It must immediately follow a section header and is followed by a period. The remainder of the section consists of zero or more procedural paragraphs to be executed when the indicated input/output error occurs. The **USE** statement is never executed; it merely defines the conditions calling for the execution of **USE** procedures. The keywords **ERROR** and **EXCEPTION** are synonymous and may be used interchangeably.

The **USE** procedure is invoked for the input/output errors specified following the optional **ON** keyword:

FILENAME	All input/output errors for the file.
INPUT	All input/output errors for files **OPEN**ed for **INPUT**.
OUTPUT	All input/output errors for files **OPEN**ed for **OUTPUT**.
I-O	All input/output errors for files **OPEN**ed for **I-O**.

The **USE** procedure is not executed when the input/output statement causing an error contains a clause, such as **AT END** or **INVALID KEY**, to handle the condition.

For additional rules concerning declaratives, see the section entitled **DECLARATIVES**.

13.4.6 WRITE Statement

```
WRITE record-name [ FROM identifier ]

    [  {BEFORE } ADVANCING { { identifier   } [  { LINES } ]  }]
       { AFTER }           { { number       }    { LINE  }    }
                           {                                  }
                           { PAGE                             }


WRITE record-name [ FROM identifier ]

       [; INVALID KEY imperative statement ]
```

The **WRITE** statement is used to release a record to an **OUTPUT** or **EXTEND** file. *To release* means that the record is conceptually transmitted to the file. The record-name is the name of a logical record used in the **FD** for the file in question.

Immediately after a file is successfully **OPEN**ed for **OUTPUT**, the logical record(s) in the **FD** is available to receive data. A **MOVE** statement, for example, may be used to move data to the logical record. A **WRITE** statement causes the record in the **FD** to be conceptually transmitted to the file. The data in that record is no longer available. Any data subsequently moved to a logical record(s) in the **FD** will be used to compose the next logical record, if any.

When the **FROM** phrase is present, the statement is equivalent to the following statements:

```
MOVE identifier TO record-name
WRITE record-name
```

when the **ACCESS** is **SEQUENTIAL** the record is written to the next position in the file. The file must be **OPEN** with for **OUTPUT**. Otherwise, the file must be **OPEN**ed for **I-O**. The position to which the record is written is determined by the value of the data item given in the **RELATIVE KEY** phrase of the **SELECT** statement. An **INVALID KEY** condition is detected when either

(1) the record already exists for the file; or

(2) an attempt is made to write a record beyond the boundaries established for the file.

The actions performed when this condition is detected are described in the section about the **READ** statement.

Chapter 14

Tables

14.1 Overview

In many computer applications it is desirable to define *tables* of data. Each element in the table has the same data description. In COBOL, this may be accomplished with the **OCCURS** clause:

```
01 cost-table.
   05 cost pic 999V99 occurs 100 times.
```

These statements illustrate how to define a table "cost" which has 100 numeric elements, each with five digits. A group item may also be repeated:

```
01 part-information.
   03 part occurs 500 times.
   05 part-number pic 9(10).
   05 cost pic 999V99.
   05 price pic 999V99.
```

This example illustrates how to specify a table "part" of 500 elements. Each element consists of three items named "part-number", "cost" and "price".

Individual elements in a table are referenced using *subscripts*:

```
cost(10)
cost(i)
```

The preceding illustration shows two examples of subscripting. The first example shows how the tenth element of "cost" is referenced. The second reference uses the value of "i" to determine which element to reference. If "i" has a value of 17, the 17-th element would be referenced.

Subscripts are written enclosed by a pair of parentheses. Each subscript is specified as a data item (not subscripted) or a numeric literal.

A subscripted data item can be used in most places that an item without subscripts might be used:

```
move cost(i) to current-cost.
add cost(i) cost(j) giving price
add sales-tax cost(j) giving bill(k)
```

The preceding examples are intended to give the "flavour" of how subscripted items might be used.

Items with a table may also be repeated with the **OCCURS** clause:

```
01  sales-data.
    03  region occurs 10 times
        05  salesman occurs 5 times.
            10 salestotal pic 9(8)V99.
            10 salescount pic 9(5).
```

The preceding example shows a table "region" of 10 elements. Each element of "region" is itself a table of 5 "salesman". Each "salesman" element consists of two items "salestotal" and "salescount". In this case, two subscripts are required to reference the elementary items:

```
salestotal(i, j)
salestotal in salesman(i, j)
salestotal in salesman in region(i, j)
```

The preceding examples illustrate three equivalent references to a "salestotal" data item for the j-th "salesman" in the i-th "region". Up to three levels of tables may be defined. Thus, it is illegal to use a **OCCURS** clause for a data item which is contained in three group items which all contain an **OCCURS** clause. A space character should follow each comma (,) character when more than one subscript is given for a data name.

The other features of COBOL table handling are:

(1) the ability to specify tables whose size varies (**OCCURS DEPENDING**); and

(2) an alternative **(INDEXING)** to subscripting as a means of referencing elements in tables.

These features are described in the detailed portions of this chapter.

14.2 OCCURS

OCCURS integer [**TO** integer] TIMES

[**DEPENDING** ON identifier]

[**INDEXED** BY index-name [, index-name] ...]

The **OCCURS** clause is used to declare a number of repeated elements of the same type. The simplest form of the clause

OCCURS integer TIMES

specifies that element is to be repeated the indicated number of times. An example of this format of the clause is given in the preceding section.

A second form of the clause may be used when the number of elements in the table is variable:

OCCURS integer **TO** integer TIMES
DEPENDING ON data-name

In this case, the number of elements in the table is determined by the value of the data item given following the **DEPENDING** keyword. The positive integer value of this data item must be in the range indicated by the positive integers following the **OCCURS** keyword. The following notes apply to this format of the **OCCURS** clause:

(1) Storage is always reserved for the maximum number of elements; the data item indicates the number of occurances of the items.

(2) No data may follow a variable-sized table in a record. Except for subordinate items, the data item containing an **OCCURS DEPENDING** clause must be the last data item in a record.

(3) This format of the **OCCURS** clause cannot be specified if the data item to which it applies is subordinate to a data item containing an **OCCURS** clause.

(4) The data name following the **DEPENDING** keyword cannot be located in the table being specified by the **OCCURS** clause.

(5) When a group item, having subordinate to it a data item with an **OCCURS DEPENDING** clause, is referenced, only that part of the table indicated by the **DEPENDING** data item will be used in the operation. Thus, variable-sized records can be read or written since only the defined part of the table is transmitted.

The Waterloo microCOBOL Interpreter will treat as an error any attempt to reference an element of a table that is beyond the bounds of the table.

The following rules apply to the **OCCURS** clause in general:

(1) The clause may not be specified for data items with level numbers 01, 66, 77 or 88.

(2) The **OCCURS** clause may also specify one or more index names. The use of these items is discussed in the following section.

14.3 Indexing

Indexing may be used as an alternative to subscripting in order the reference elements in a table. Subscripts are integer values, presented as a numeric literal or a data item. Index values are contained in either index-names (specified by **INDEXED** phrase of **OCCURS** clause) or in index data items (data items with a **USAGE IS INDEX** clause). The normal arithmetic calculations of COBOL are used to assign integer values to data items used a subscripts. Index values are assigned to index-names or index data items using the **SET or PERFORM** statements.

Indexing is intended to provide an efficient mechanism to access elements in a table. The index values are "hidden" from the COBOL programmer; they may be implemented in whatever manner is efficient for the hardware on which the COBOL program executes.

Index data items are used only to store index values. Indexing is accomplished only with index names and/or with numeric literals. Consider the following COBOL statements:

```
02  COST OCCURS 100 TIMES
          INDEXED BY COST-INDEX
          PICTURE 9(8)V99.
.....
SET  COST-INDEX TO 47.
MOVE   49.34 TO COST (COST-INDEX).
```

The **SET** statement causes the appropriate index value to reference the 47-th item of "COST" to be assigned to the index-name "COST-INDEX". The next statement illustrates how "COST" can be indexed using this index name. The **MOVE** statement would cause the value 49.34 to be assigned to the 47-th item of "COST".

The following terms may be used as an index:

index-name
index-name + literal
index-name - literal
literal

where the literal is a positive numeric literal. As with subscripts, up to three indices may be required depending upon the number of tables to which a data item is subordinate. The general form of indexing is:

data-name (index, [, index [, index]])

The next section describes the **SET** statement which may be used place values in index names or index data items.

14.4 SET Statement

```
SET  { identifier    [, { identifier   } ] ... } TO   { identifier   }
     { index-name       { index-name   }              { index-name   }
                                                      { number       }

SET  index-name [, index-name ] ...       { UP BY     } { identifier  }
                                          { DOWN BY   } { number      }
```

The **SET** statement is used to assign values to index-names or to index-data items. It may also be used to assign an integer value, representing the number of the element in the table being referenced by a index name to a data item.

When the **TO** clause is present, the **SET** statement is used to assign a value representing a position in a table. There are four possibilities for the item following the **TO** keyword:

(1) elementary data item which is an integer: the value of the data item represents the position in a table.

(2) elementary data item whose **USAGE** is **INDEX**: the value of the data item indicates a position in any table.

(3) index name: the value of the index name represents a position in the table which defined that index name.

(4) numeric literal: the value of the literal is the table position.

The value representing this position is assigned to each of the items following the **SET** keyword. There are three possibilities for each of these items:

(1) integer data item: this item may receive only an integer representing the position indicated by an index name.

(2) index name: this item may receive a value representing a position in the table for which it is defined, from any of the possibilities following the **TO** keyword.

(3) index data item: this item may receive only a value representing a position in any table from either another index data item or from an index name.

Only the possibilities outlined above are permitted.

When the **UP BY** or **DOWN BY** clause is used, the values of index names following the **SET** keyword are adjusted relatively by a number of positions in the table for which they are each defined. The number of positions to be adjusted is given by the value of the integer literal or of the elementary integer data item:

```
SET COST-INDEX UP BY 2
```

The example shows how an index data item can be adjusted two onward in a table. Thus, if "**COST-INDEX**" indicated the 47-th position in the table before the **SET** statement was executed, it would indicate the 49-th position after execution of the statement.

Chapter 15

String Manipulation

15.1 Overview

Three verbs are provided to manipulate data as strings:

INSPECT	provides the capability to count and/or replace occurrances of characters in a data item.
STRING	provides the capability to compose part or all of a data item from a number of strings.
UNSTRING	provides the capability to extract the contents of parts of a data item and assign these parts to other data items.

The **INSPECT** and **UNSTRING** verbs are often useful for scanning data which is free-format and/or variable sized. Any COBOL data item may be used as a string. The data items are viewed as sequences of characters to be manipulated using the string verbs. The **STRING** verb is often useful for constructing output which is not aligned upon field boundaries.

15.2 INSPECT Statement

```
INSPECT identifier TALLYING

  {,identifierFOR      {,  { { ALL           } { identifier  } }
                       {   { { LEADING       } { literal     } }
                       {   {  CHARACTERS                       }

               [ { BEFORE  } INITIAL  { identifier  } ] } ... } ...
                 { AFTER   }          { literal     }


INSPECT identifier REPLACING

   {CHARACTERSBY { identifier  }
   {             { literal     }
   {
   {{,{ALL          } {, identifier  } BY { identifier  }
   {{ {FIRST        } { literal      }    { literal     }
   {{ {LEADING      }

               [ { BEFORE  } INITIAL  { identifier  } ] } ... } ... }
                 { AFTER   }          { literal     }


INSPECT identifier TALLYING

  {,identifierFOR      {,  { { ALL     } { identifier  } }
                       {   { { LEADING } { literal     } }
                       {   {  CHARACTERS                   }

               [ { BEFORE  } INITIAL  { identifier  } ] } ... } ...
                 { AFTER   }          { literal     }
```

```
REPLACING

{CHARACTERSBY { identifier  }
{               { literal     }
{
{{,{ALL         } {, identifier  } BY { identifier  }
{{ {LEADING     } { literal      }      { literal     }
{{ {FIRST       }

        [ { BEFORE } INITIAL { identifier } ] } ... } ... }
          { AFTER  }         { literal    }
```

The **INSPECT** statement provides the capability to count and/or replace occurrances of groups of characters in a data item. The **TALLYING** clause specifies the character groups to be counted, the conditions under which they are counted, and the data item to contain the count. The **REPLACING** clause specifies the character groups to be replaced, the replacement values, and the conditions under which replacement takes place. When both clauses are present the statement is treated as if it were two **INSPECT** statements, the first with an identical **TALLYING** clause and the second with an identical **REPLACING** clause.

Both the **TALLYING** and **REPLACING** clauses specify a number of character-group occurrances for which to search. The comparison cycle proceeds as follows:

(1) The comparison starts with the first character in the data item following the **INSPECT** keyword.

(2) The character groups are processed, in order specified in the **TALLYING** or **REPLACING** clause, searching for the first one to match the data item starting with the current character position.

    (a) If no match is found, the comparison position is advanced by one.

    (b) If a match is found, a **TALLYING** or **REPLACING** operation is performed and the comparison position is advanced by the size of the matched item.

(3) The preceding step is repeated provided the entire data item has not been inspected. Otherwise, the execution of the statement is completed.

Thus, the comparison cycle proceeds a character at a time until a match is found. The comparison resumes, following a match, at a position adjusted onward by the size of the item matched.

Each of the **TALLYING** or **REPLACING** phrases may contain a **BEFORE** or **AFTER** keyword to restrict the range over which the comparison cycle actively considers the phrase. When the **BEFORE** keyword is given, the phrase is actively considered only up to the character immediately preceding the character string given as a literal or data item following that keyword. When the **AFTER** keyword is given, the phrase is actively considered only following the last character of the character string given as a literal or data item following that keyword. If neither phrase is specified, the phrase is actively considered throughout the inspected data item.

The character string to be matched by the comparison cycle may be specified in a number of ways:

(1) **CHARACTERS** : this is a one-character item which matches any character in the data item being inspected.

(2) **ALL** data-name or literal : the character string to be matched is the value of *the literal or data item.*

(3) **LEADING** data-name or literal : the character string to be matched is the value of the literal or data item ; the match is valid only at the first position for which the clause is to be actively considered, when a match occurs, each of the contiguous occurances of the matched string in the data item is counted/replaced.

(4) **FIRST** data item or literal : the character string to be matched is the value of literal or data item; the clause is no longer actively considered after it has been successfully matched.

15.3 STRING Statement

```
STRING  { identifier } [,  { identifier } ] . . .
        { literal    }     { literal    }

     DELIMITEDBY { identifier }
                 { literal    }

      [ { identifier } [,  { identifier } ] . . .
        { literal    }     {literal     }

     DELIMITEDBY { identifier } ] . . .
                 { literal    }

     INTO identifier [ WITH POINTER identifier ]

     [; ON OVERFLOW imperative statement ]
```

The **STRING** statement is used to place one or more "small" strings of characters into a "large" data item. The placement can start (**POINTER** phrase) anywhere in the "large" string. For each of the "small" strings, a delimiting character string may be given to specify only the portion of the string up to the delimiter are to be placed into the "large" string.

The data item following the **INTO** keyword is the "large" string into which the "small" strings are placed. The placement starts at the leftmost character of the data item when the **POINTER** phrase is not specified. When the **POINTER** phrase is specified, the data item following that keyword must contain a positive integer value used as the offset (one represents the leftmost position of the data item) at which the placement will start. This data item is incremented by one each time a character is placed into the receiving data item. It may be noted that multiple **STRING** statements, using the **POINTER** data item, may be used to construct single "large" data item. The **POINTER** data item will contain the offset used in the next **STRING** statement to place its characters immediately following those placed by the preceding **STRING** statement.

Preceding the **INTO** keyword are given a number of sequences of data items or literals, each followed by a **DELIMITED** phrase. Each of the data items or literals are considered in the order they are given in the **STRING** statement, the portion of these "small" strings placed into the "large" string depends upon the first **DELIMITED** phrase which follows the character string:

(1) **SIZE**: the entire character string is placed in the "large" character string.

(2) literal or data item: only the portion of the "small" character string up to, but not including, the value of the delimiting literal or data item is placed in the "large" character string.

The placement of characters into the "large" character string proceeds a character at a time. The process is completed when either

(1) the "small" character strings have all been moved to the "large" string; or

(2) the value of the **POINTER** data item is non-positive or large than the size of the "large" string.

In the latter case, the imperative statement associated with the **OVERFLOW** clause will be executed, if this clause is specified.

15.4 UNSTRING Statement

```
UNSTRING identifier

[   DELIMITED BY [ ALL ]  { identifier  }
                          { literal     }

         [ OR [ ALL ]  { identifier  } ] ... ]
                       { literal     }

    INTOidentifier

        [, DELIMITER IN identifier ]

        [, COUNT IN identifier ]

      [,   identifier

        [, DELIMITER IN identifier ]

        [, COUNT IN identifier ] ] ...

    [ WITH POINTER identifier ]

    [ TALLYING IN identifier ]

    [; ON OVERFLOW imperative statement ]
```

The **UNSTRING** verb may be used to create "small" strings from a "large" string. The "large" string is given by the data item immediately following the **UNSTRING** keyword. The **UNSTRING** process may start anywhere (**POSITION** phrase) in the "large" string. The number of "small" strings created may be counted using the **TALLYING** keyword. Each "small" string (named following an **INTO** keyword) is created from characters in the "large" string, starting at the current position and continuing to either the end of the string or to the point immediately preceding the a delimiting character string (specified in a **DELIMITING** phrase). For each such receiving string, the specific delimiter encountered may be saved (**DELIMITER IN**) as may be the number of characters to be moved to the receiving string (**COUNT IN**) The current position is advanced following each movement to the next position to the right of the delimiting character string in the "large" string.

When the **POINTER** keyword is not given, the "large" string is processed starting at the leftmost character of the string. When the keyword is present, the value of the data item following the keyword is used as an offset (one represents the leftmost position), in order to establish the point at which processing starts. At the completion of the statement, the **POINTER** data item will contain the offset of the next unexamined character in the "large" string. Thus, another **UNSTRING** statement may then be executed with this **POINTER** value to continue the **UNSTRING**ing process at the point completed by the initial **UNSTRING** statement.

The **DELIMITED BY** phrase is used to give one or more character sequences to be used to delimit the characters to be moved to the current "small" data item. When the **ALL** keyword is present, multiple occurrances of the delimiter value (given by the data item or literal following the keyword) are treated as if the value occurred once. Multiple delimiters may be given by separating the specifications with the **OR** keyword.

When the **TALLYING** keyword is present, the value one is added to the data item specified following the keyword, each time a "small" string has data moved to it. In this way, a count of the number of **UNSTRING** operations can be maintained. The **UNSTRING** statement does not initialize this data item in any way.

When the **OVERFLOW** clause is present, the imperative statement given in that clause is executed under the following conditions:

(1) the data item given following the **POINTER** keyword is non-positive or greater than the size of the "large" character string; or

(2) all the "small" items have been processed and there still exist unexamined characters in the "large" string.

15.5 Formatting Example

In order to illustrate some of the features of string manipulation, a sample program has been included in this section. The program reads a file (unformatted text) of 80-character records and produces another file (formatted text) of 80-character records. An input record is composed of zero or more words separated from one another by one or more space characters.

The program scans words and adds them to an output line. When the addition of a word would exceed the capacity of a line, that line is written and the word is added to the start of the next line. Thus, the program may be considered to be a primitive text formatting program.

```
*
* word/line problem
*
 identification division.
 program-id. WORDLINE.
 environment division.
 configuration section.
 source-computer. IBM-4331.
 object-computer. IBM-4331.

 input-output section.

 file-control.
     select optional card-file
         assign to 'unfmt'
         file status is card-status.
     select line-file
         assign to 'fmted'.

 data division.

 file section.

 fd  card-file
     label records are standard.
 01  card-record.
     02 filler          pic x(80).
```

```
fd   line-file
     label records are standard.
01   line-record.
     02 filler          pic x(80).

working-storage section.

77   card-status         pic xx.

77   card-ptr           pic 99.
77   line-ptr           pic 99.
77   word-size          pic 99.
77   word-count         pic 99.

77   got-word          pic xxxx.

77   card-data         pic x(80).
77   line-data         pic x(80).
77   word-data         pic x(20).

procedure division.

     open input card-file.
     perform init-line.
     move '00' to card-status
     perform read-card.
     perform process-card
          until card-status not equal '00'.
     perform fini-line.
     close card-file.
     stop run.

read-card.
     read card-file into card-data
          at end.
     display card-data.

process-card.
     move 1 to card-ptr.
     perform get-word.
     perform process-word
          until got-word not equal 'true'.
     perform read-card.
```

```
get-word.
     move zero to word-count.
     move spaces to word-data.
     unstring card-data
          delimited by all space
          into word-data
               count in word-size
          with pointer card-ptr
          tallying word-count.
     if word-count greater than zero
          move 'true' to got-word
          add 1 to word-size
     else
          move 'nope' to got-word.
     display 'word:'word-data.

process-word.
     if word-size + line-ptr greater than 81
          perform write-line
          perform new-line.
     string
          word-data delimited by space
          space delimited by size
          into line-data
          with pointer line-ptr.
     perform get-word.

init-line.
     open output line-file.
     perform new-line.

new-line.
     move 1 to line-ptr.
     move spaces to line-data.

fini-line.
     perform write-line.
     close line-file.

write-line.
     write line-record from line-data.
     display line-data.
```

Chapter 16

Interactive Debugger

16.1 Overview

The interactive debugger is an integral part of the microCOBOL interpreter system. It is designed to be used to monitor the execution of a program. The facilities provided include the capability to execute COBOL statements immediately, to execute statements in the program one at a time and to continue or terminate execution. The debugger is entered when an error occurs during the execution of a program, when the BREAK key (or an equivalent key) is depressed, or when an **ENTER DEBUGGING** statement is executed.

It should be noted that the microCOBOL interpreter will check the syntax of the entire program before the actual execution of the program is commenced. Any syntax errors detected at this point do not cause the debugger to be entered. The debugger is entered only after the actual execution of the program has started.

When the debugger is entered, a number of messages are displayed at the terminal. These messages show the sections or paragraphs which are being actively performed at the time of the error. In addition, the statement in error is displayed, with an indication of the position in the statement at which the error was detected. A full English-text error message is also displayed.

Debugging commands are entered as single letters, optionally followed by extra information. The following sections describe these commands.

16.2 Continue (c) Command

The continue command causes the microCOBOL interpreter to resume execution of the COBOL program starting with the current statement. This command is typically used following an **ENTER DEBUGGING** statement or after the user has replaced a data value which caused the error to be detected.

The statement at which execution resumes is the one following the last one executed, unless the debugger was entered because of an error. In the latter case, execution resumes with the statement that caused the error.

16.3 Execute (e) Command

```
e sentence
```

The Execute command causes a COBOL sentence to be executed, as if the sentence were inserted into the program (followed by an **ENTER DEBUGGING** statement) at the point in the program at which the debugger was entered.

The debugger is normally re-entered, in the same state as existed before the sentence was executed, after successful execution of the sentence. An exception to this rule is the successful execution of a **GO TO** statement. In this situation the debugger is terminated and execution continues normally at the target statement.

When an error occurs while executing the sentence, the debugger is not entered recursively. It is re-entered with same state as existed before the sentence was began execution with the Execute command. Thus, the suspended statement is the one at which the debugger was originally invoked.

The Execute statement has many powerful uses when debugging programs. The contents of data items may be inspected by executing a **DISPLAY** statement:

```
e display myvar
```

The preceding example causes the value of the data item "myvar" to be displayed upon the screen. A section or paragraph may be executed by executing a **PERFORM** statement. Values may be placed into data items by executing **MOVE** statements:

```
e move 79.34 to amount.
```

The preceding example illustrates how the value 79.34 may be placed in the data item "amount".

Sometimes an error may be temporarily corrected by executing one or more statements. For example, an attempt to use an undefined value might be corrected by executing a **MOVE** statement to place an appropriate value in the data item. It would then be possible to use the Continue (c) command to resume execution of the program. In other cases, the debugger should be terminated with the Quit (q) command.

16.4 Quit (q) Command

The Quit command causes the execution of the program to terminate and the editing subsystem to be re-entered.

16.5 Step (s) Command

The Step command causes the program to execute the single statement at which the debugger has suspended execution. Depressing the RETURN key another time causes the next statement to execute. In some implementations, keeping the RETURN key depressed causes the program to execute with each line to be executed displayed immediately before it is executed. Thus, the flow of control can be precisely viewed.

16.6 Where-am-I (w) Command

The Where-am-I command causes the messages displayed, when the debugger was initially entered, to be displayed again on the terminal. The command may be used to remind a user where the program is suspended and of the error that caused the debugger to be entered.

16.7 ENTER DEBUGGING

ENTER DEBUGGING ENVIRONMENT

The **ENTER DEBUGGING** statement is used to enter the debugging at points specified by the programmer. This statement is an extension to standard COBOL is intended to be used only when debugging programs using the microCOBOL interpreter.

Chapter 17

CALL Statement

```
CALL { identifier } USING identifier
     { literal    }

  [, { identifier } ] ...
     { literal    }
```

The **CALL** statement, as implemented, is an *extension* to COBOL. It is intended to be used only to invoke machine-language subroutines. Waterloo microCOBOL provides no support for the Inter-Program Communication module described in the COBOL language.

The integer data item or literal following the **CALL** keyword is used as the address of the subroutine to be invoked. The integer data item following the **USING** keyword contains a return value, if any, from that subroutine. The remaining data items or literals are passed to the invoked subroutine as parameters.

The method by which parameters and return values are communicated with the called subroutine is dependent upon the computer system on which the COBOL program executes (see SYSTEM DEPENDENCIES). In general, the convention used is compatible with that used by the WSL (Waterloo Systems Language) programming language.

Chapter 18

System Dependencies

18.1 Overview

System dependencies arise because the hardware and controlling programs differ from computer system to computer system. A COBOL processor will, in general, buffer the user from many of these dependencies. In some cases, however, it is better that a programmer be aware of these dependencies in order that a program is able to execute on various systems.

System dependencies are most often encountered in the following areas:

(1) file system

(2) collating sequence

(3) hardware constraints of peripherals

(4) calling assembly-language subroutines

These issues are discussed in each of the system dependent sections. As well, a section on portability is included to act as a guide for those who wish to execute programs on multiple computing systems.

18.2 Portability

Essentially, portability is the ability to move a program from one computing environment to another. The amount of effort this entails is a measure of the degree of portability of the program in question. A number of techniques can be used to increase the portability of a program.

18.2.1 File Names

The file naming conventions differ from system to system. However, most systems support short file names (say 6 characters) composed of uppercase letters.

18.2.2 Use of Files

Files should be used in the most straight-forward way possible. Techniques to be avoided include:

(1) creating a file with **SEQUENTIAL** organization and then processing it with **RELATIVE** organization.

(2) extending the size of a file with **RELATIVE** organization.

(3) creating a **RELATIVE** file with **RANDOM** access.

These techniques work on many systems, but not all. It is often expensive to reprogram an application which uses one or more of these capabilities.

18.2.3 Code Set

Waterloo microCOBOL supports only the native (hardware) implementation of the collating sequence. There are two principal code sets in popular use; *EBCDIC* (larger IBM computers) and *ASCII* (most other computers). Since characters are arranged differently in these code sets, writing programs to depend upon a specific ordering should be avoided.

18.3 Commodore SuperPET

This section outlines the system dependencies for the Commodore SuperPET. More detail is found in the System Overview Manual for that computer.

18.3.1 Code Set

The Commodore SuperPET uses the ASCII collating sequence.

18.3.2 Date Support

In order that the **ACCEPT FROM DATE** verb produce the correct result, the current date (see **DATE** command in **EDITOR** description) should be set as "YYMMDD" where "YY" is the last two digits of the year, "MM" is the number of the month, and "DD" is number of the day. Thus, September 25, 1983 is entered as "830925".

18.3.3 Files

Files are completely described in the System Overview manual. This section deals with the aspects that pertain directly to microCOBOL. There are two formats of files which may be stored on Commodore 2040 or 8050 diskettes, "seq" and "rel". These files correspond, roughly, to the COBOL sequential and relative organizations.

A file name, on a diskette system, is given in the format:

```
(type:size)device:name[,        { seq  } ]
                                { rel  }
```

where "name" is given as up to 16 characters, including special characters and spaces. If omitted, "seq" is assumed.

Because "seq" files may be processed only in a sequential manner, they should be used only with **ORGANIZATION IS SEQUENTIAL**. When **ORGANIZATION IS RELATIVE**, "rel" files must be used since only these files permit random access.

There are three types of files which may be stored in either of the two formats:

text	A text file consists of variable-sized records, containing only "printable" characters. This file type is chosen by default when the type is not mentioned.
variable	A variable file consists of variable-sized records which may contain arbitrary characters.
fixed	A fixed file consists of fixed-sized records which may contain arbitrary characters.

Fixed files should be used when all records in the file have the same size; otherwise, variable or text files should be used. Text files should not be used to store files in which there are index data items or signed numeric values in which the **SIGN IS SEPARATE** is not given.

The *size* is the maximum size, in characters, of a record in the file. For fixed files, this size is the size of all records.

The file system with the SuperPET does not store file type or size information. Consequently, each time a file is used, the type, size and format specifications should be given as part of the file name. The safest convention is to use the identical file specification each time the file is mentioned.

18.3.4 Listing Files

When the **ADVANCING** keyword is used with a **WRITE** statement, it must be used for all **WRITE** statements for that file. Record descriptions should reserve an extra character at the start of each record for carriage-control information. This character is filled in automatically by microCOBOL.

When the **ADVANCING** keyword is not used for a file and that file is recognizable as a listing file, the first character in each record written is assumed to be a character used for vertical positioning. The only such file recognized on the SuperPET is the file "printer".

The control characters '1', '0', '+', '-', and ' ' are translated to ASCII form-feed, line-feed and carriage-return characters, or combinations of characters, automatically by microCOBOL. Where large numbers of lines are skipped, blank records may be written to the file to ensure proper vertical spacing.

18.3.5 Call Interface

The execution of the **CALL** statement causes an assembly-language subroutine to be invoked. The parameters, if any, given in the **CALL** statement are passed to the invoked routine as follows:

integer	An integer data item or literal is passed as a two-byte binary value.
other	The data item or literal is copied to a temporary location and a byte with hexadecimal zeroes is appended to the end of the copied value. The address of the temporary copy is passed to the assembly-language routine.

All parameters are passed upon the stack pointed at by the SP register. The address to which the assembly-language routine should return is pushed on the stack following the parameters, if any. The address to which control is passed is obtained by taking the integer value given following the **CALL** keyword and treating that value as an address.

The assembly-language subroutine should return to the address pushed at the top of the SP stack. When that return takes place, that address should have been popped from the stack. The parameters should still reside upon the stack. The contents of the hardware D register are used as the return value. This value is placed in the data item, if present, given following the **USING** keyword.

Consider the following example:

```
77    ADDR PIC 99999.
77    INT-VAL PIC 99.
77    CHR-VAL PIC X(5).
77    RET-VAL PIC 9(5).
. . .
      MOVE 27 TO INT-VAL.
      MOVE "ABCDE" TO CHR-VAL.
      MOVE 21346 to ADDR.
      CALL ADDR USING RET-VAL, INT-VAL, CHR-VAL.
      DISPLAY RET-VAL.
```

The execution of the **CALL** statement will cause a temporary copy of "CHR-VAL" to be placed in memory (say at location 19437). A zeroed byte is appended following the five characters "ABCDEF" at this location. The contents of the SP stack when the routine at location 21346 receives control are as follows, (all entries are two-byte values):

(top)	16457	(return address)
	27	(value of INT-VAL)
(bottom)	19437	(address of temporary string)

When the assembly-language subroutine returns to address 16457, at the completion of its execution, the stack contents will appear:

(top)	27	(value of INT-VAL)
(bottom)	19437	(address of temporary string)

If the hardware D register contains 963, then that value is placed into "RET-VAL". Consequently,

```
00963
```

will be **DISPLAY**ed by the statement following the **CALL** statement in the example.

Notes:

(1) It is the responsibility of the programmer to load the assembly-language subroutine into memory and to supply the correct address of that routine.

(2) The library routines described in *Waterloo 6809 Assembler : Tutorial and Reference Manual* may be called using the CALL statement.

(3) The Waterloo 6809 WSL compiler generates subroutines in 6809 assembly language which may be invoked with the **CALL** statement.

18.4 VM/CMS

This section outlines the system dependencies for the IBM VM/CMS operating system.

18.4.1 Code Set

The computers on which VM/CMS executes use the EBCDIC collating sequence.

18.4.2 Files

File names in the VM/CMS file system are given as

name type mode

and are described completely in the documentation written by IBM for this operating system. Generally, users will specify only the *name* and occasionally the *type*. These names may be up to 8 characters in length and composed of letters and digits.

When creating files it is not necessary to specify any information about the size of records or the format of the files. This information is automatically determined by microCOBOL.

There is no difference between files organized sequentially and randomly. Carriage control characters are the normal EBCDIC characters '1', '0', ' ', '-' and '+'. Where large numbers of lines are **ADVANCED**, blank lines may be inserted in the file.

18.4.3 Listing Files

When the **ADVANCING** keyword is used with a **WRITE** statement, it must be used for all **WRITE** statements for that file. Record descriptions should reserve an extra character at the start of each record for carriage-control information. This character is filled in automatically by microCOBOL.

When the **ADVANCING** keyword is not used for a file and that file is recognizable as a listing file, the first character in each record written is assumed to be a character used for vertical positioning. The files recognized in the VM/CMS are the file "printer" and files with a type of "LISTING".

The control characters '1', '0', '+', '-', and ' ' are not translated to any other character as most IBM printers use these characters for vertical spacing. Where large numbers of lines are skipped, blank records may be written to the file to ensure proper vertical spacing.

18.4.4 Call Interface

The execution of the **CALL** statement causes an assembly-language subroutine to be invoked. The parameters, if any, given in the **CALL** statement are passed to the invoked routine as follows:

integer	An integer data item or literal is passed as a four-byte binary value.
other	The data item or literal is copied to a temporary location and a byte with hexadecimal zeroes is appended to the end of the copied value. The address of the temporary copy is passed to the assembly-language routine.

All parameters are passed using a list pointed at by register 12. The address to which the assembly-language routine should return is contained in register 14. The address to which control is passed is obtained by taking the integer value given following the **CALL** keyword and treating that value as an address.

The assembly-language subroutine should return to the address contained in register 14. The contents of the register 11 are used as the return value. This value is placed in the data item, if present, given following the **USING** keyword.

Consider the following example:

```
77   ADDR PIC 99999.
77   INT-VAL PIC 99.
77   CHR-VAL PIC X(5).
77   RET-VAL PIC 9(5).
. . .
     MOVE 27 TO INT-VAL.
     MOVE "ABCDE" TO CHR-VAL.
     MOVE 21346 to ADDR.
     CALL ADDR USING RET-VAL, INT-VAL, CHR-VAL.
     DISPLAY RET-VAL.
```

The execution of the **CALL** statement will cause a temporary copy of "CHR-VAL" to be placed in memory (say at location 19437). A zeroed byte is appended following the five characters "ABCDEF" at this location. Register 12 points at a list as follows:

27	(value of INT-VAL)
19437	(address of temporary string)

If register 11 contains 963 when the assembly-language subroutine completes execution, then that value will be placed in "RET-VAL" and

```
00963
```

will be **DISPLAY**ed by the statement following the **CALL** statement in the example.

Notes:

(1) It is the responsibility of the programmer to load the assembly-language subroutine into memory and to supply the correct address of that routine.

(2) The Waterloo VM/CMS WSL compiler generates subroutines in /370 assembly language which may be invoked with the **CALL** statement.

Appendix A

Language Skeleton

This appendix gives the skeleton for the syntax accepted by Waterloo microCOBOL. It is organized by division.

A.1 IDENTIFICATION DIVISION.

A.1.1 Skeleton

IDENTIFICATION DIVISION.

PROGRAM-ID. name.

[**AUTHOR.** [comment]]

[**INSTALLATION.** [comment]]

[**DATE-WRITTEN.** [comment]]

[**DATE-COMPILED.** [comment]]

[**SECURITY.** [comment]]

A.2 ENVIRONMENT DIVISION

A.2.1 Skeleton

```
   ENVIRONMENT DIVISION.

   CONFIGURATION SECTION.

   SOURCE-COMPUTER. name [WITH DEBUGGING MODE ].

   OBJECT-COMPUTER. name.

                                   { WORDS      }
      [,MEMORY SIZE number  { CHARACTERS }]
                                   { MODULES    }

      [,PROGRAM COLLATING SEQUENCE is name ]

 [ SPECIAL-NAMES.

      [,CURRENCY SIGN IS literal ]

      [,DECIMAL-POINT IS COMMA ] ].

 [ INPUT-OUTPUT SECTION.

   FILE-CONTROL.

      {select clause} . . . ]
```

A.2.2 SELECT Clause

```
SELECT [ OPTIONAL ] file-name

    ASSIGN TO literal

    [; ORGANIZATION IS { RELATIVE   }]
                       { SEQUENTIAL }

    [; ACCESS MODE IS  { SEQUENTIAL [,RELATIVE KEY IS name ]  }]
                       {                                      }
                       { {RANDOM  } , RELATIVE KEY IS name    }
                       { {DYNAMIC }                           }

    [; FILE STATUS IS name ].
```

A.3 DATA DIVISION

A.3.1 Skeleton

```
DATA DIVISION.

[ FILE SECTION.

[ FD filename

    (FD entry)

    (record-description entry) . . . ] . . . ]

[ WORKING-STORAGE SECTION.

{ 77 (data-description)          } . . . ]
{ (record-description entry)     }
```

A.3.2 FD entry

```
[; BLOCK contains [ number TO ] number         { RECORDS     }]
                                               { CHARACTERS  }

[; RECORD CONTAINS [ number TO ] number CHARACTERS ]

[; LABEL {RECORD IS      } { STANDARD  } ]
         {RECORDS ARE    } { OMITTED   }

[; VALUE OF literal is literal

[; DATA { RECORD IS      } name, name ... ]
        { RECORDS ARE    }

[; CODE-SET IS name ]
```

A.3.3 Data-description entry: Level 66

```
66 name-1; RENAMES name-2 [   { THROUGH  } name-3 ]
                              { THRU     }
```

A.3.4 Data-description entry: Level 88

```
88 name; { VALUE IS    } literal      [ { THROUGH  } literal ]
         { VALUES ARE  }              [ { THRU     }

         [ , literal [          { THROUGH   } literal ] ] . . .
                                { THRU      }
```

A.3.5 Data-description entry: Levels 01-49

```
level-number   { data-name  }
               { FILLER     }

    [; REDEFINES data-name ]

    [; { PICTURE  } IS character string ]
       {PIC       }

                       { COMPUTATIONAL  }
    [; [ USAGE IS ]    { COMP           } ]
                       { DISPLAY        }
                       { INDEX          }

    [; SIGN IS ]   { LEADING  } [ SEPARATE CHARACTER ]
                   {TRAILING }

    [; OCCURS   { number TO number TIMES DEPENDING on name } ]
                { number TIMES }

        [ INDEXED BY name [, name ....] ]

    [;  { SYNCHRONIZED  } [  { LEFT   } ] ]
        { SYNCH         }    { RIGHT  }

    [;   { JUSTIFIED  } RIGHT ]
         { JUST       }

    [; BLANK WHEN ZERO ]

    [; VALUE is literal ] .
```

A.4 PROCEDURE DIVISION

A.4.1 Skeleton

```
    PROCEDURE DIVISION.
[   DECLARATIVES.
{   section-name SECTION. declarative sentence
[   paragraph-name. [ sentence ] ... ] } ...
    END DECLARATIVES. ]
    (procedure body)
```

A.4.2 Procedure Body

```
{   paragraph-name. [ sentence ] . . . }
```

or

```
{   section-name SECTION.
[   paragraph-name. [ sentence ] ... ] ... } ...
```

A.4.3 Statements

```
ACCEPT identifier [ FROM   { DATE } ]
                           { TIME }


ADD  { identifier } [,  { identifier } ] ...
     { literal    }     { literal    }

   TO identifier [ ROUNDED ] [, identifier [ ROUNDED ] ...

   [; ON SIZE ERROR imperative statement ]


ADD  { identifier }, { identifier } [,  { identifier } ] ...
     { literal    }  { literal    }     {literal       ]

   GIVING identifier [ ROUNDED ] [ identifier [ ROUNDED ] ]...

   [; ON SIZE ERROR imperative statement ]


ADD  { CORRESPONDING  } identifier TO identifier [ ROUNDED ]
     { CORR           }

   [; ON SIZE ERROR imperative statement ]


ALTER procedure-name TO [ PROCEED TO ] procedure-name

   [, procedure-name TO [ PROCEED TO ] procedure-name ] ...


CALL { identifier } USING identifier
     { literal    }

  [,   { identifier } ] ...
       { literal    }
```

```
CLOSE file-name [  { { REEL } [  { WITH NO REWIND   } ]  } ]
                   { { UNIT }    { FOR REMOVAL      }    }
                   {                                     }
                   { WITH  { NO REWIND }                 }
                   {       { LOCK      }                 }

   [ , file-name ... ] ...
```

```
COMPUTE identifier [ ROUNDED ] [, identifier [ ROUNDED ] ] ...

   = arithmetic-expression

   [; ON SIZE ERROR imperative statement ]
```

```
DISPLAY  { identifier } [,  { identifier } ] ...
         { literal    }     { literal    }
```

```
DIVIDE  { identifier } INTO identifier [ ROUNDED ]
        { literal    }

   [, identifier [ ROUNDED ] ] ...

   [; ON SIZE ERROR imperative statement ]
```

```
DIVIDE  {identifier } INTO { identifier }
        { literal   }      {literal     }

   GIVING identifier [ ROUNDED ] [, identifier [ ROUNDED ] ]...

   [; ON SIZE ERROR imperative statement ]
```

```
DIVIDE  { identifier } BY { identifier }
        { literal    }    { literal    }

   GIVING identifier [ ROUNDED ] [, identifier [ ROUNDED ] ]...

   [; ON SIZE ERROR imperative statement ]
```

```
DIVIDE  { identifier } INTO { identifier }
        { literal    }      { literal    }

   GIVING identifier [ ROUNDED ] REMAINDER identifier

   [; ON SIZE ERROR imperative statement ]


DIVIDE  { identifier } BY { identifier }
        { literal    }    { literal    }

   GIVING identifier [ ROUNDED ] REMAINDER identifier

   [; ON SIZE ERROR imperative statement ]


ENTER DEBUGGING ENVIRONMENT


EXIT


GO TO [ procedure-name ]


GO TO procedure-name [, procedure-name ] ...

   DEPENDING ON identifier


IF condition;   { statement      } [; ELSE { statement      }]
                { NEXT SENTENCE  }         { NEXT SENTENCE  }


INSPECT identifier TALLYING

   {,identifierFOR    {, { { ALL          } { identifier } }
                      {  { { LEADING      } { literal    } }
                      {  {  CHARACTERS                   }

             [ { BEFORE } INITIAL { identifier } ] } ... } ...
               { AFTER  }         { literal    }
```

```
INSPECT identifier REPLACING

    {CHARACTERSBY { identifier  }
    {             { literal     }
    {
    {{,{ALL      } {, identifier  } BY { identifier  }
    {{ {FIRST    } { literal      }    { literal     }
    {{ {LEADING  }

          [ { BEFORE } INITIAL { identifier } ] } ... } ... }
            { AFTER  }         { literal    }


INSPECT identifier TALLYING

    {,identifierFOR   {, { { ALL     } { identifier } }
                      {  { {LEADING } { literal    } }
                      {  { CHARACTERS               }

          [ { BEFORE } INITIAL { identifier } ] } ... } ...
            { AFTER  }         { literal    }

    REPLACING

    {CHARACTERSBY { identifier  }
    {             { literal     }
    {
    {{,{ALL      } {, identifier  } BY { identifier  }
    {{ {LEADING  } { literal      }    { literal     }
    {{ {FIRST    }

          [ { BEFORE } INITIAL { identifier } ] } ... } ... }
            { AFTER  }         { literal    }


MOVE  { identifier } TO { identifier } [, identifier ] ...
      { literal    }


MOVE  { CORRESPONDING } identifier TO identifier
      { CORR          }
```

```
MULTIPLY { identifier } BY  { identifier } [ ROUNDED ]
         { literal    }

    [, identifier [ ROUNDED ] ] . . .

    [; ON SIZE ERROR imperative statement ]


MULTIPLY { identifier } BY { identifier }
         { literal    }    { literal    }

    GIVING identifier [ ROUNDED ]

         [, identifier [ ROUNDED ] ] . . .

    [; ON SIZE ERROR imperative statement ]


OPEN { INPUT  } file-name, [, file-name ] ...
     { OUTPUT }
     { I-O    }
     { EXTEND }

   [ { INPUT  } file-name, [, file-name ] ... ] ...
     { OUTPUT }
     { I-O    }
     { EXTEND }


PERFORM procedure [ { THROUGH } procedure ]
                    { THRU    }

    [ { identifier } TIMES ]
      { number     }


PERFORM procedure [ { THROUGH } procedure ]
                    { THRU    }

    [ UNTIL condition ]
```

```
PERFORM procedure [   { THROUGH  } procedure ]
                      { THRU     }

    VARYING  { identifier } FROM { identifier   }
             { literal    }      { index-name   }
                                 { literal      }

    BY  { identifier } UNTIL condition
        { literal    }

    [ AFTER  { identifier   } FROM { identifier   }
             { index-name   }      { index-name   }
                                   { literal      }

    BY  { identifier } UNTIL condition
        { literal    }

    [ AFTER  { identifier   } FROM { identifier   }
             { index-name   }      { index-name   }
                                   { literal      }

    BY  { identifier } UNTIL condition ] ]
        { literal    }

READ file-name [ NEXT RECORD] [ INTO identifier ]

    [; AT END imperative statement ]

READ file-name RECORD [ INTO identifier ]

    [; INVALID KEY imperative statement ]

REWRITE record-name [ FROM identifier ]

    [; INVALID KEY imperative statement ]

SET  { identifier    [, { identifier    } ] ... } TO    { identifier   }
     { index-name       { index-name    }               { index-name   }
                                                        { number       }
```

```
SET    index-name [, index-name ] ...        { UP BY      } { identifier   }
                                             { DOWN BY    } { number       }


STOP   { RUN     }
       { literal }


STRING  { identifier } [,  { identifier } ] . . .
        { literal    }     { literal    }

     DELIMITEDBY { identifier }
                 { literal    }

      [ { identifier } [,  { identifier } ] . . .
        { literal    }     {literal     }

     DELIMITEDBY { identifier } ] . . .
                 { literal    }

     INTO identifier [ WITH POINTER identifier ]

     [; ON OVERFLOW imperative statement ]


SUBTRACT  { identifier } [,  { identifier } ... ]
          { literal    }     { literal    }

     FROM identifier [ ROUNDED ] [ identifier [ ROUNDED ] ]...

     [; ON SIZE ERROR imperative statement ]
```

```
SUBTRACT  { identifier } [,  { identifier } ... ]
          { literal    }     {literal      }

    FROM { identifier }
         { literal    }

    GIVING identifier [ ROUNDED ]

        [, identifier [ ROUNDED ] ] ...

    [; ON SIZE ERROR imperative statement ]


SUBTRACT  { CORRESPONDING  } identifier
          { CORR           }

    FROM identifier [ ROUNDED ]

    [; ON SIZE ERROR imperative statement ]
```

```
UNSTRING identifier

 [   DELIMITED BY [ ALL ]  { identifier  }
                           { literal     }

         [ OR [ ALL ]  { identifier  } ] ... ]
                       { literal     }

    INTOidentifier

        [, DELIMITER IN identifier ]

        [, COUNT IN identifier ]

      [,  identifier

        [, DELIMITER IN identifier ]

        [, COUNT IN identifier ] ] ...

    [ WITH POINTER identifier ]

    [ TALLYING IN identifier ]

    [; ON OVERFLOW imperative statement ]


USE AFTER STANDARD  { EXCEPTION  } PROCEDURE
                    {  ERROR     }

    ON { file-name [, file-name ] ...        }
       { INPUT                               }
       { OUTPUT                              }
       { I-O                                 }
       { EXTEND                              }


WRITE record-name [ FROM identifier ]

  [   {BEFORE } ADVANCING { { identifier   } [  { LINES  } ]  }]
      { AFTER }           { { number       }    { LINE   }    }
                          {                                   }
                          { PAGE                              }
```

WRITE record-name [**FROM** identifier]

[; **INVALID** KEY imperative statement]

Appendix B

Reserved Words

The following is a list of reserved words in the full COBOL language. Waterloo microCOBOL treats all the words as reserved, even though many are not required in the current language definition. This ensures compatibility with other COBOL processors.

ACCEPT	CH	DATE-WRITTEN	FOR
ACCESS	CHARACTER	DAY	FROM
ADD	CHARACTERS	DE	EMI
ADVANCING	CLOCK-UNITS	DEBUG-CONTENTS	ENABLE
AFTER	CLOSE	DEBUG-ITEM	END
ALL	COBOL	DEBUG-LINE	END-OF-PAGE
ALPHABETIC	CODE	DEBUG-NAME	ENTER
ALSO	CODE-SET	DEBUG-SUB-1	ENVIRONMENT
ALTER	COLLATING	DEBUG-SUB-2	EOP
ALTERNATE	COLUMN	DEBUG-SUB-3	EQUAL
AND	COMMA	DEBUGGING	ERROR
ARE	COMMUNICATION	DECIMAL-POINT	ESI
AREA	COMP	DECLARATIVES	EVERY
AREAS	COMPUTATIONAL	DELETE	EXCEPTION
ASCENDING	COMPUTE	DELIMITED	EXIT
ASSIGN	CONFIGURATION	DELIMITER	EXTEND
AT	CONTAINS	DEPENDING	FD
AUTHOR	CONTROL	DESCENDING	FILE
BEFORE	CONTROLS	DESTINATION	FILE-CONTROL
BLANK	COPY	DETAIL	FILLER
BLOCK	*CORR*	*DISABLE*	*FINAL*
BOTTOM	CORRESPONDING	DISPLAY	FIRST
BY	COUNT	DIVIDE	FOOTING
CALL	CURRENCY	DIVISION	FOR
CANCEL	DATA	DOWN	FROM
CD	DATE	DUPLICATES	GENERATE
CF	DATE-COMPILED	FOOTING	GIVING

GO	MEMORY	PROGRAM-ID	SENTENCE
GREATER	MERGE	QUEUE	SEPARATE
GROUP	MESSAGE	QUOTE	SEQUENCE
HEADING	MODE	QUOTES	SEQUENTIAL
HIGH-VALUE	MODULES	RANDOM	SET
HIGH-VALUES	MOVE	RD	SIGN
I-O	MULTIPLE	READ	SIZE
I-O-CONTROL	MULTIPLY	RECEIVE	SORT
IDENTIFICATION	NATIVE	RECORD	SORT-MERGE
IF	NEGATIVE	RECORDS	SOURCE
IN	NEXT	REDEFINES	SOURCE-
INDEX	NO	REEL	COMPUTER
INDEXED	NOT	REFERENCES	SPACE
INDICATE	NUMBER	RELATIVE	SPACES
INITIAL	NUMERIC	RELEASE	SPECIAL-NAMES
INITIATE	OBJECT-COMPUTER	REMAINDER	STANDARD
INPUT	OCCURS	REMOVAL	STANDARD-1
INPUT-OUTPUT	OF	RENAMES	START
INSPECT	OFF	REPLACING	STATUS
INSTALLATION	OMITTED	REPORT	STOP
INTO	ON	REPORTING	STRING
INVALID	OPEN	REPORTS	SUB-QUEUE-1
IS	OPTIONAL	RERUN	SUB-QUEUE-2
JUST	OR	RESERVE	SUB-QUEUE-3
JUSTIFIED	ORGANIZATION	RESET	SUBTRACT
KEY	OUTPUT	RETURN	SUM
LABEL	OVERFLOW	REVERSED	SUPPRESS
LAST	PAGE	REWIND	SYMBOLIC
LEADING	PAGE-COUNTER	REWRITE	SYNC
LEFT	PERFORM	RF	SYNCHRONIZED
LENGTH	PF	RH	TABLE
LESS	PH	RIGHT	TALLYING
LIMIT	PIC	ROUNDED	TAPE
LIMITS	PICTURE	RUN	TERMINAL
LINAGE	PLUS	SAME	TERMINATE
LINAGE-COUNTER	POINTER	SD	TEXT
LINE	POSITION	SEARCH	THAN
LINE-COUNTER	POSITIVE	SECTION	THROUGH
LINES	PRINTING	SECURITY	THRU
LINKAGE	PROCEDURE	SEGMENT	TIME
LOCK	PROCEDURES	SEGMENT-LIMIT	TIMES
LOW-VALUE	PROCEED	SELECT	TO
LOW-VALUES	PROGRAM	SEND	TOP

TRAILING
TYPE
UNIT
UNSTRING
UNTIL
UP
UPON
USAGE
USE
USING
VALUE
VALUES
VARYING
WHEN
WITH
WORDS
WORKING-STORAGE
WRITE
ZERO
ZEROES
ZEROS

Waterloo microCOBOL©

Waterloo microCOBOL is a substantial implementation of the standard COBOL language and is suitable for teaching purposes and for the programming of many business problems. The language includes many features described in COBOL Standards ANSI X3.23-1974 and ISO 1989-1978.

This book is divided into two sections. In the first part, a collection of annotated examples to introduce the reader to microCOBOL is given. Examples include implementation of:

- Introductory examples
- Reading and writing files
- Arithmetic
- Printing & editing numeric values
- Subscripted data names
- Relative files
- and more

The second section is a detailed reference manual describing the language supported by Waterloo microCOBOL. Waterloo microCOBOL is implemented in a number of different compiler systems. While most of this manual applies to all implementations, a chapter on System Dependencies is also included to describe features particular to a specific system. Items covered include:

- The four divisions of Waterloo microCOBOL programs — IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION.
- Discussions of the various statements used in microCOBOL and explanations of their use.
- Complete explanations of Sequential Files, Relative Files, Tables and String Manipulation — and their use.
- The use of the interactive Debugger in monitoring program execution.

DISTRIBUTED BY

Howard W. Sams & Co., Inc.

4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

$9.95/21909

ISBN: 0-672-21909-3