IBM

SYSTEM/360 COBOL

COBOL Programming Techniques

Text

Programmed Instruction Course

IBM

SYSTEM/360 COBOL

COBOL Programming Techniques

Text

Programmed Instruction Course

Copies of this publication can be obtained through IBM Branch Offices.
Address comments concerning the contents of this publication to:
IBM DPD Education Development, Education Center, Endicott, New York

PREFACE

The general objective of this book is to teach students techniques by which they can make the most effective and efficient use of COBOL. Major emphasis is centered on discussions of the ways in which fundamental procedural statements are executed, and what the programmer can do to cause the most efficient execution of these statements. Students learn that the instructions compiled for a statement like "ADD A, B, GIVING C" may cause data to be not only added, but also moved, converted to a different data code, shifted to align decimals, truncated, and edited, to name only a few possibilities! These actions depend partly on the way in which the programmer has defined the data items and specified the processing to be done; students learn how to write entries so as to eliminate unnecessary actions, in order to get efficient processing while still getting the desired data results.

The statements which are discussed are: First, the MOVE statement and the three ways in which it may be executed -- alphanumeric moves, numeric moves, and edit moves; included are the rules for using all editing picture characters. Second, arithmetic statements, including the ROUNDED and SIZE ERROR options. Third, IF statements, emphasizing compound conditional expressions, nested IF statements, and the execution of relation tests. Fourth, PERFORM statements -- especially the "TIMES", "UNTIL", and "VARYING" options.

Throughout, the text explains how these fundamental statements may be applied to typical programming situations. In addition, one lesson is devoted to ways of writing program switches in COBOL, and two lessons deal with data tables and subscripting.

To teach COBOL facts, rules, and techniques is not the only aim of this book, however. Another aim is to develop good working habits -- concern for object program efficiency, and attention to details, for instance. A final, and very important, aim is to make the student self-sufficient in his future work with COBOL; to this end, the student is given many reading assignments in the regular COBOL reference manual, so that he will know how to use it to find whatever information he may need in the future.

This is the third in a series of programmed instructions courses on System/360 COBOL. The student is expected to have completed the previous courses: COBOL Program Fundamentals (text R29-0205 and reference handbook R29-0206), and Writing Programs in COBOL (text R29-0210 and reference handbook R29-0211).

ACKNOWLEDGEMENT

The following information is reprinted from COBOL-61 EXTENDED, published by the conference on Data Systems Languages (CODASYL), and printed by the U. S. Government Printing Office.

This publication is based on the COBOL System developed in 1959 by a committee composed of government users and computer manufacturers. The organizations participating in the original development were:

Air Materiel Command,
United States Air Force
Bureau of Standards,
Department of Commerce
David Taylor Model Basin,
Bureau of Ships, U.S. Navy
Electronic Data Processing Division, Minneapolis-Honeywell Regulator Company
Burroughs Corporation
International Business Machines Corporation
Radio Corporation of America
Sylvania Electric Products, Inc.
Univac Division of Sperry-Rand Corporation

In addition to the organizations listed above, the following organizations participated in the work of the Maintenance Group:

Allstate Insurance Company
Bendix Corporation, Computer Division
Control Data Corporation
DuPont Company
General Electric Company
General Motors Corporation
Lockheed Aircraft Corporation
National Cash Register Company
Philco Corporation
Royal McBee Corporation
Standard Oil Company (N.J.)
United States Steel Corporation

This manual is the result of contributions made by all of the above-mentioned organizations. no warranty, express or implied, is made by any contributor or by the committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

It is reasonable to assume that a number of improvements and additions will be made to COBOL. Every effort will be made to insure that the improvements and corrections will be made in an orderly fashion, with due recognition of existing users' investments in programming. However, this protection can be positively assured only by individual implementors.

Procedures have been established for the maintenance of COBOL. Inquiries concerning procedures and methods for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.

TABLE OF CONTENTS

STUDENT INSTRUCTIONS

1. This is the third in a series of programmed instruction courses on System/360 COBOL. The preceding courses (COBOL Program Fundamentals and Writing Programs in COBOL) are prerequisites to this course.

2. Be sure to read the Preface of this book, which explains the overall goal of this course.

3. Besides this book, you must have:

   - a System/360 COBOL reference manual. (Use the manual that corresponds to the operating system your installation uses. For the full Operating System, use IBM Operating System/360 COBOL Language, Form C28-6516. For Disk or Tape Operating System, use IBM System/360 Disk and Tape Operating System, COBOL Language Specifications, Form C24-3433.

   - a pad of COBOL program sheets (Form X28-1464).

   - the reference handbooks you received in the earlier courses of this series (Form R29-0206 and R29-0211).

4. All reading assignments given in this textbook are in your reference manual. An assignment will name the topic you are to read, but will not give the page numbers in the manual; you are expected to look up the topic in the Index of the manual to get the page numbers for yourself. Also, you should use the reference handbooks from the earlier courses whenever you need to review what you studied before. In a few cases, this text will ask you to read specific topics in the reference handbooks.

5. This textbook will be used by other students after you, so do not fill in any of the blanks or make any notes in this book.

6. The format of this book is exactly the same as that used in the previous texts of this series. As before, topics of study are presented in a series of frames, with most frames requiring you to choose an answer or to formulate an answer mentally. The correct answers are given right after each question. You should use a card or a sheet of paper to cover up the correct answer until you have formulated your own answer to a question.

7. If the meanings of symbols like brackets and braces (as they are used in frames) are fresh in your mind, you may begin Lesson 1; otherwise, read "How to Study this Book" on the next page.

HOW TO STUDY THIS BOOK

1. Each lesson is broken up into a number of frames, which are simply convenient instructional steps to be studied in sequence. Most frames have two parts: the first part usually asks a question or requires you to take some action; the second part gives the correct answer to the question. The end of the first part is marked by a group of three dots. If the frame asks a question, the correct answer is printed on the same page, below the three dots.

2. As you study each frame, you must use an ordinary sheet of paper or a card to hide the correct answer from yourself. You will learn best by working out the answers, not by just reading words.

3. Start each page by putting your "hider" sheet or card at the top. Then slide your sheet down until you uncover a group of three dots. This allows you to read the first part of a frame, and to formulate your answer to the question or problem it poses. When you have your answer clearly in mind, slide the "hider" sheet down to the next group of three dots. This reveals the correct answer, and also uncovers the first part of the next frame. Frames vary in length; a page may contain only one frame, or as many as seven frames.

4. Your answer to a frame may sometimes be different from the printed answer, but it should mean the same. If your answer is wrong, study the question again with the correct answer in mind.

5. On the whole, the course is composed of reading assignments and questions. When a frame gives you a reading assignment, be sure to complete the reading before you go on to the next frame. The frames that follow a reading assignment may ask questions about what you have read, or ask you to apply what you have read; they may also provide additional information about the topic. You will find instructions, remarks, and the author's opinions printed in italics in some frames.

6. When you come to a blank _____ in a frame, you are to think of one or more words that complete the sentence. The length of the blank space is always the same, so it is not a clue to the length of the answer. Do not write your answer in the book.

7. Some frames present a choice of answers, from which you are to select the one best answer. The choices are stacked in braces { }.

8. Other frames present a choice of answers, from which you are to select all correct answers. All of the choices, more than one choice, just one choice, or none of the choices may be correct. It is therefore necessary for you to examine every choice. Each choice of this kind is enclosed in brackets [].

LESSON 1

1 *You already know how to cause data to be moved from one place to another in storage. However, you will be able to do an even more effective job of using MOVE statements when you have learned what happens during a move operation. You will learn that there are really three types of moves, and that each type involves a large number of different actions. Lessons 1, 2, and 3 are devoted to this subject.*

●●●

2 *The first piece of information that is useful to know is that certain moves are not permitted! That is, you are not allowed to move data from certain items to certain other items. Your reference manual contains a table of all moves, showing which moves are permitted and which are not; take a couple of minutes to examine the table, but don't try to memorize the moves that are permitted. In succeeding frames we will discuss why some moves are permitted while others are not, and we will try to simplify the table of permissible moves for study purposes.*

Reading assignment: MOVE statement -- Table of permissible moves

To find this table, look under "MOVE" in the Index of your reference manual. Only examine the table at this time; other information about the MOVE statement will be covered by later reading assignments.

●●●

3 By actual count, there are about equal numbers of Yeses and Noes in the table you have just looked at. Why is it that about half of all possible moves are not legal? A large part of the answer to this question lies in the definitions of the classes of items. For instance, items classed as "alphabetic" are permitted to contain only letters and spaces, whereas items that are "numeric" (external decimal, internal decimal, binary, and floating-point) may contain only digits.

This would explain why alphabetic source items { can / cannot } be moved to numeric receiving items.

●●●

cannot

4 At the same time, it is plain to see why you are not allowed to move numeric source items to alphabetic receiving items. On the other hand, items classed as "alphanumeric" may contain all kinds of characters -- letters, digits, special characters, or spaces.

You would expect, then, that [numeric] [alphabetic] items can be moved to alphanumeric items.

●●●

BOTH numeric AND alphabetic

In fact, all except floating-point items (which have a special kind of format) can be moved to alphanumeric receiving items.

5 Here is a simplified version of the table in the reference manual. Only the main kinds of items are included in this version; figurative constants and sterling (British currency) items are omitted. Also, all numeric items have been lumped together here, since they are treated alike (with a minor exception in the case of floating-point items). The shaded boxes represent moves that are not valid.

VALID MOVES

		Receiving Item				
			Elementary			
Source Item		Group	Alphanumeric	Alphabetic	Numeric	Report
Group		✓	✓	✓		
Elementary	Alphanumeric	✓	✓	✓		
	Alphabetic	✓	✓	✓		
	Numeric	✓	✓*		✓	✓
	Report	✓	✓			

* Whole numbers only, and not floating-point numbers

●●●

6 COBOL students sometimes jump to the conclusion that if it is legal to move one item to another (say, item A to item B), it is also legal to move the items in the other direction (item B to item A). Is this conclusion correct?

●●●

No. For example, although it is legal to move a numeric item to an alphanumeric item, it is not legal to move an alphanumeric item to a numeric item. You can find several other examples as well.

7 Our simplified table makes it easier to see the patterns of valid and invalid moves. It is especially easy to see that only _____ source items can be moved to numeric receiving items.

●●●

numeric

You may wish to look at the larger table in the reference manual to see for yourself that this is an accurate rule.

8 A "report" item is used to receive numeric data which will be edited with spaces or special characters when the data is moved into the item. If the number 12345 were moved to a report item, the report item might contain $123.45 after the move.

Can you explain why a report item must not be moved to:

1. an alphabetic item?
2. a numeric item?
3. another report item?

●●●

1. Alphabetic items are not allowed to contain digits and special characters.
2. Numeric items are not allowed to contain special characters.
3. Report items can receive only numeric data which has not yet been edited; moving from one report item to another would mean editing data that is already edited.

9 We have been dealing with the four main classes of data items: alphanumeric, alphabetic, numeric, and report. If you look back at the table of valid moves, you will find that an item's class must be considered

{ regardless of whether it is a group item or an elementary item.
only if it is a group item.
only if it is an elementary item. }

●●●

only if it is an elementary item

10 Can you recall what the difference is between group and elementary items?

●●●

A group item is made up of smaller items. An elementary item is not made up of smaller items. Or stated another way, in the Data division a group item is further subdivided, while an elementary item is not further subdivided.

11 Our simplified table indicates that all moves are valid in which a group item is the [source item] [receiving item].

●●●

receiving item

12 Besides items, we frequently use literals in MOVE statements. Here are two examples, the first containing a numeric literal, the second a non-numeric literal.

```
    MOVE 532 TO MINIMUM-LEVEL.
    MOVE '90 DAYS OVERDUE' TO ACTION.
```

While literals are not shown on the tables of permissible moves, all literals are treated as elementary items. Numeric literals are treated as numeric items. Non-numeric literals are treated as

{ alphanumeric items.
alphabetic items.
report items. }

●●●

alphanumeric items

This is because non-numeric literals, like alphanumeric items, may contain all kinds of characters.

13 Let's apply this information about valid and invalid moves to a practical case. Suppose that you are checking a program, and you come across this statement in the Procedure division:

```
MOVE DATE TO INVOICE-DATE.
```

Can you tell by looking at this statement whether or not it is a valid move?

●●●

No, you must find out the level (group or elementary) and class (alphanumeric, alphabetic, numeric, or report) of each item.

14 You could find this additional information in the _____ division of the program.

●●●

Data

15 Carrying this example further, suppose that these are the Data division entries that describe DATE and INVOICE-DATE:

```
    03  DATE.
        04  MONTH, PICTURE 99.
        04  DAY, PICTURE 99.
        04  YEAR, PICTURE 99.

02  INVOICE-DATE, PICTURE 9(6).
```

From these entries, we learn that both DATE and INVOICE-DATE contain numeric data. We also learn that

{ DATE and INVOICE-DATE are both elementary items.
DATE is an elementary item, but INVOICE-DATE is a group item.
DATE is a group item, but INVOICE-DATE is an elementary item.
DATE and INVOICE-DATE are both group items. }

●●●

DATE is a group item, but INVOICE-DATE is an elementary item

(Remember that items with pictures must be elementary items.)

16 You can now definitely conclude that the sample move (MOVE DATE TO INVOICE-DATE) { is / is not } a valid move.

You may refer to the table of permissible moves if you wish.

●●●

is not

A group item cannot be moved to an elementary numeric item.

17 *We will have a little more to say about this particular example later in this lesson. For now, let's sum up the point it teaches: A move that "looks reasonable" is not necessarily valid! Whenever you are in doubt, refer to the table of permissible moves.*

●●●

18 You have also seen that it is quite important to be able to identify the class of an elementary item. Most of the time, we can do this by looking at the picture of the item. To refresh your memory on how to identify an item from its picture, you may wish to read the discussion of PICTURE clause, under Item Description Entries (Data Division), in the COBOL Program Fundamentals reference handbook.

●●●

19 Whenever a picture contains the letter A, you can tell that the item is { alphanumeric / alphabetic } .

●●●

alphabetic

20 Pictures of alphanumeric items always contain the letter _____.

●●●

X

21 Pictures of numeric items contain the digit 9 and may also contain S (sign), V (assumed decimal point), and P (assumed position).

Pictures of report items resemble those of numeric items, in that they too may contain the characters 9, V, and P. However, a report item's picture must also contain one or more editing symbols, such as

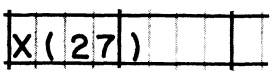
```
{ (1)   # @ % & }
{ (2)   " ? ; ! }
{ (3)   $ . * + }
```

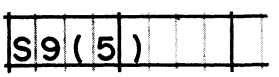
●●●

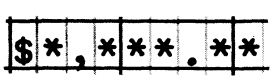
(3) $. * +

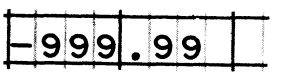
22 Identify the class of each item described by the pictures below.

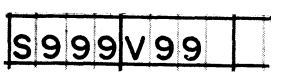
(1) 

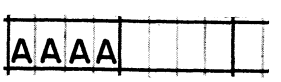
(2) 

(3) 

(4) 

(5) 

(6) 

(7) 

(8) 

●●●

(1) report
(2) alphanumeric
(3) numeric
(4) report
(5) report
(6) numeric
(7) alphabetic
(8) alphanumeric

23 *Different classes of items are moved in different ways. The reference manual briefly outlines some of the main differences. While the manual identifies two types of "simple" moves, we will find it convenient to think in terms of three types; you will learn about this shortly.*

Reading assignment: MOVE statement

Note 1: Here is a tip on how to read the reference manual. Read it twice! The first time, read through the material fairly rapidly, just to get a general idea of what the book is talking about. The second time, pay closer attention to the facts and rules which the book presents.

Note 2: If your reference manual contains information about two options of the MOVE statement, read about Option 1 only. Option 2 (the "MOVE CORRESPONDING" option) applies to special situations, and can be used only when the source computer has larger than average storage capacity; therefore, we will not discuss Option 2 in this book. I believe that after you have mastered the "simple" MOVE (Option 1), Option 2 will seem equally simple -- and if necessary you will be able to learn about it easily on your own.

●●●

24 *Let me summarize the general idea of what you have just read in the manual: The actions that occur during a move vary somewhat, depending on the kind of source item and the kind of receiving item. In this course, we will talk about three types of moves: alphanumeric, numeric, and edit moves.*

●●●

25 **To help you visualize when each type of move occurs, here is our table of valid moves again, this time showing the types of moves:**

TYPES OF MOVES

A = Alphanumeric
N = Numeric
E = Edit

Source Item		Receiving Item: Group	Receiving Item: Elementary			
		Group	Alphanumeric	Alphabetic	Numeric	Report
Group		A	A	A		
Elementary	Alphanumeric	A	A	A		
	Alphabetic	A	A	A		
	Numeric	A	A*		N	E
	Report	A	A			

*** Whole numbers only, and not floating-point numbers**

●●●

26 **The type of move that occurs in nearly all permissible moves is _____.**

●●●

alphanumeric

27 **A numeric move takes place when [a group numeric item] [an elementary numeric item] is moved to [a group numeric item] [an elementary numeric item].**

●●●

ONLY when an <u>elementary</u> numeric item is moved to an <u>elementary</u> numeric item.

All group items are treated as alphanumeric, even though they may contain numeric data.

28 An edit move occurs only when an _____ item is moved to an _____ item.

●●●

elementary numeric to elementary report

29 *We will study each type of move in detail. The A's dominate the table, so it seems reasonable to begin with the alphanumeric type of move. Happily, this type also involves the fewest actions -- and therefore the fewest rules to learn. The remainder of this lesson is devoted to the rules that govern the action of alphanumeric moves. Numeric moves are the subject of Lesson 2, and edit moves are discussed in Lesson 3.*

●●●

30 Alphanumeric move. There are just three rules:

Rule 1. Left justification. Data is aligned at the left end of the receiving item.

Rule 2. Receiving item longer than source item. Any extra positions at the right end of the receiving item are filled with spaces (blanks).

Rule 3. Receiving item shorter than source item. After the receiving item is filled, no more characters are moved from the source item.

●●●

31 After an alphanumeric move, the data in the receiving item will look exactly like the data in the source item, provided that

{ the receiving item is shorter than the source item.
the source item is shorter than the receiving item.
the receiving item is the same length as the source item. }

●●●

the receiving item is the same length as the source item.

32 Suppose that the value of the source item is TFX and the picture of the receiving item is A(5). After the move, the value of the receiving item will be _____.

●●●

TFXbb *(Throughout this book, little b's are used to represent blanks -- spaces -- in the value of an item.)*

33 Assume that the picture of a receiving item is X(6). What will be the value of this item following a move if the value of the source item is

(1) KANSAS
(2) OHIO
(3) bOHIO

●●●

(1) KANSAS
(2) OHIObb
(3) bOHIOb

"Left justification" does not include elimination of left-hand blanks. The initial blank in "bOHIO" is regarded as the first character in the source item, and it is moved to the leftmost position of the receiving item.

34 The contents of a receiving item before a move do not affect the actions that occur during the move.

Say that the value 96823 is moved to an item whose picture is X(8). Which choice below correctly shows the contents of the item after the move -- (1) if the previous value of the item was 00000000 and (2) if the previous value was 12345678?

(1) 96823000 and (2) 96823678
(1) 00096823 and (2) 12396823
(1) 96823bbb and (2) 96823bbb
(1) 96823000 and (2) 96823000

●●●

(1) 96823bbb and (2) 96823bbb

35 When the receiving item is not large enough to hold all of the characters from the source item, the extra characters are not moved. We say that the remaining characters in the source item are <u>truncated</u>.

"Truncate" means { "to cut off" / "to exchange" / "to erase" }

●●●

"to cut off"

Truncation does not change the source item's value in any way. (The contents of the source item are not altered by any type of move.)

36 If the name SMITH were moved to an item whose picture is AA, which characters would be truncated?

●●●

ITH

Note that neither the compiler nor the computer considers truncation to be an error. Both assume that the programmer knows what he is doing. In the example given in this frame, presumably the programmer wants only the first two letters of a name for some reason.

37 Take a case in which BEARINGSbbbb is the value of the source item. On a piece of scratch paper, jot down the value of the receiving item after the move, if the receiving items picture is:

(1) X(6)
(2) X(12)
(3) X(18)

●●●

(1) BEARIN
(2) BEARINGSbbbb
(3) BEARINGSbbbbbbbbb

38 *An alphanumeric move can be "reversed" if desired. That is, you can cause the data to be right-justified (aligned at the right end of the receiving item). Accordingly, truncation or filling with blanks then takes place at the left end of the data.*

Such a reversed move will occur if the description of the receiving item (in the Data division) includes a JUSTIFIED RIGHT clause.

Reading assignment: JUSTIFIED RIGHT clause

●●●

39 *It is not likely that you will find much use for the right-justified move, so we will not spend much time on it. I think I owe you one example of where you might be able to use such a move, though.*

Imagine that you are preparing a customer bill, at the bottom of which you want to print three lines that look like this:

```
GROSS AMOUNT    $205.30
      DISCOUNT    $4.11
NET AMOUNT      $201.19
```

In this case, the words that identify the amounts are "justified right". This result can be accomplished in more than one way, of course. Using the JUSTIFIED RIGHT clause is an easy way to do it.

We begin by defining a single item to which the words will be moved. This item description would be part of a record description. We make the item large enough to contain the longest string of characters (GROSSbAMOUNT). And we specify that data moved to this item is to be right-justified.

```
02  AMOUNT-ID, PICTURE A(12),
    JUSTIFIED RIGHT.
```

At appropriate points during processing, these procedural statements are acted on:

```
MOVE 'GROSS AMOUNT' TO AMOUNT-ID.

MOVE 'DISCOUNT' TO AMOUNT-ID.

MOVE 'NET AMOUNT' TO AMOUNT-ID.
```

When the literals are moved to the item, they will be right-justified.

●●●

40 Another programmer, faced with the problem described in the preceding frame, defined AMOUNT-ID like this:

```
02 AMOUNT-ID, PICTURE A(12).
```

He then wrote these MOVE statements:

```
MOVE 'GROSS AMOUNT' TO AMOUNT-ID.
MOVE '  DISCOUNT' TO AMOUNT-ID.
MOVE '  NET AMOUNT' TO AMOUNT-ID.
```

Was this programmer's technique a correct solution to this problem?

●●●

Yes

In fact, this solution is probably the better one, because the programmer himself has right-justified the literals once and for all, instead of having the computer do it each time the job is run.

41 You will recall that the normal alphanumeric move does not eliminate left-hand (leading) blanks. Similarly, the reversed alphanumeric move does not eliminate right-hand (trailing) blanks.

To illustrate, suppose that a receiving item is defined as JUSTIFIED RIGHT. Its picture is X(10). What will this item contain after a move if the value of the source item is:

(1) CARROTSbbb
(2) PEASbb

●●●

(1) CARROTSbbb
(2) bbbbPEASbb

LESSON 2

42 *This lesson deals with the numeric type of move. The numeric move is significantly different from the alphanumeric move. The main difference is that the numeric move is concerned about the meaning of the data it transfers, whereas the alphanumeric move doesn't bother about the data's meaning.*

Let me explain. To the alphanumeric move, data consists of a string of bytes, each byte containing a bunch of bits. Those bits have a meaning that depends on the data code; a string of bits means one thing if the data code is binary, and something completely different if the code is packed-decimal. In COBOL terms, items can have different usage. However, the alphanumeric move transfers data without regard for the usage of the items. Thus, if a computational (binary) item were moved to an alphanumeric display (BCD) item, the data would not be converted from one data code to the other.

By contrast, if the binary item had been moved to a numeric display item, a numeric move would have been performed, and the data would have been converted from binary to BCD. The numeric move pays attention not only to the usage of the source and receiving items, but also to their signs and the locations of their decimal points.

●●●

43 Numeric move.

Rule 1. Data code conversion. The data is converted, if required, to match the usage of the receiving item.

Since conversion depends entirely upon the usage of the source item and the receiving item, it would be wise to review what usage means.

Reading assignment: USAGE clause

You may also find it useful to re-read the information on the USAGE clause in the COBOL Program Fundamentals reference handbook. You will find it under Item Description Entries in the section on the Data division.

●●●

44 Translate the COBOL usage terms below into the terms that are normally used to describe the data codes of the System/360.

(1) COMPUTATIONAL-3
(2) COMPUTATIONAL
(3) DISPLAY

●●●

(1) COMPUTATIONAL-3: "packed decimal" or "internal decimal"
(2) COMPUTATIONAL: "binary"
(3) DISPLAY: "BCD" (binary-coded decimal), "EBCDIC" (extended binary-coded decimal interchange code), or "external decimal"

45 Conversion is done by instructions generated by the compiler, and usually takes place in a work area set aside by the compiler. There is a price to be paid for conversion, then, in terms of time and storage space. When possible, conversion should be avoided.

Which of the statements below is the best rule for avoiding conversion during a move?

- Make sure that the lengths of the source and receiving items are equal.
- Try to keep the usage of the source and receiving items the same.
- Do not allow the source and receiving items to have unlike signs.

●●●

Try to keep the usage of the source and receiving items the same.

46 Keeping the usage of items the same is good advice if you are designing the formats of data records. But suppose that the records have already been designed, and your job is just to get them processed. Sooner or later you may run into a situation like this: an input record contains a packed decimal item which is to be moved to four BCD items at various points in a procedure. If you simply write four moves, the same item is going to be converted from internal to external decimal four times!

You can avoid such unnecessary conversions by moving the data first to an intermediate area whose usage is the same as that of the final receiving fields. Then move from the intermediate area to the receiving fields. In this way, the data is converted only once, at the time it is moved to the intermediate item.

For the situation described above, which of these entries defines an intermediate item with the desired usage?

(1)
```
77  BOX-1, PICTURE S9(7).
```

(2)
```
77  BOX-1, PICTURE S9(7), USAGE DISPLAY.
```

(3)
```
77  BOX-1, PICTURE S9(7), COMPUTATIONAL.
```

(4)
```
77  BOX-1, PICTURE S9(7), COMPUTATIONAL-3.
```

●●●

Both (1) and (2) are acceptable, since they define external decimal items.

47 *Conversion is only one of the actions that may go on during a numeric move. The remaining actions depend on the pictures of the source and receiving items. Before we go on to additional numeric move rules, read what the manual has to say about pictures of numeric items.*

Reading assignment: PICTURE clause, Numeric-form option

●●●

48 Numeric move.

Rule 2. Operational signs. If the receiving item's picture contains an S, a sign is developed. Otherwise, bits that represent "no sign" are put into the receiving item.

●●●

49 The option of either developing a sign or developing "no sign" bits applies to external decimal (BCD) and packed decimal items. It does not apply to binary items. Why not?

●●●

Binary (COMPUTATIONAL) items always have a sign. In their case, there is no such thing as a "no sign" bit-configuration. Therefore, an S is always required in the pictures of binary items.

50 To develop "no sign" bits requires the compiler to generate additional instructions in the object program. Therefore, it is best to have an S in the picture, except where the receiving item definitely must not have a sign.

You probably would not want the item to have a sign if its contents are to be [used as a factor in an arithmetic calculation] [printed on a report] [written on magnetic tape] [tested by an IF statement].

●●●

printed on a report

When a signed number is printed, the sign bits are treated as the zone bits of an alphabetic character, so a letter rather than a digit is printed in the units position. (It is possible to get a sign indicator to print -- for instance, a plus sign or minus sign, "CR" or "DB" -- but this is one of the functions of the edit move which you will study later.) You might also want to have no sign if the data were punched in a card, where the sign would result in a zone overpunch in the low-order position.

51 Numeric move.

Rule 3. Alignment. The assumed decimal point in the value of the source item is aligned with the assumed decimal point in the receiving item.

●●●

52 *In numeric items, decimal points are always assumed. They are not stored as actual characters in the item. Because the letter V is used to show the location of assumed decimal points in pictures, we will use a little "v" in this book to show the assumed decimal points within data values, for example "1v375".*

Where is the assumed decimal point located? If there is a V in an item's picture, such as S999V99, then the answer is obvious. But suppose there is no V, as in the pictures shown below. For each of these pictures, tell where the assumed decimal point is.

(1) S9999
(2) SPPP9(8)
(3) S99P(5)

●●●

(1) This is a whole number (integer). The assumed decimal point is at the end of the number. The picture of this item might also have been written as S9999V.
(2) The Ps represent "assumed positions" and are used to locate the assumed decimal point away from the actual number. So in this example, the assumed decimal point is located in front of the first P. The picture might also have been written as SVPPP9(8).
(3) The assumed decimal point is located after the last P. The picture might have been written as S99P(5)V or S99PPPPPV.

53 *In this book, to represent the situations discussed above, we will use little "p's" for assumed positions. For instance, if an item's picture is PPP999, and the item actually contains the digits 235, we will show the item's value as "vppp235". When it comes to integers, we will show their value without a "v"; for example, "50" and not "50v".*

●●●

54 Numeric literals must enter our thinking too. Unless they are integers, literals are written on program sheets with real decimal points, like .2635, 4.1796, etc. It is important to realize that these decimal points are not stored as separate characters when the literals are put into items in storage.

So the value of the literal 3.1416 is really _____, and the literal .75 is treated as _____.

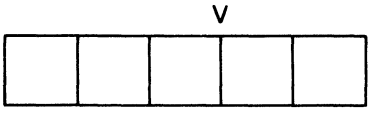
●●●

3v1416; v75

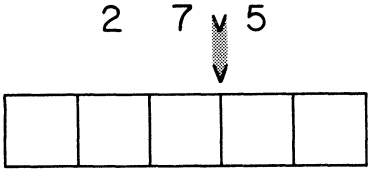
55 Numeric move. When the decimal points are aligned.

Rule 4. Extra positions in receiving item. Any extra positions, at either or both ends of the receiving item, are filled with zeros.

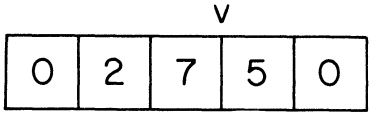
We can visualize this action graphically. Suppose that the picture of the receiving item is 999V99. We can show this item as a string of five boxes:



Suppose that the source value is 27v5. The assumed decimal point of this value is aligned with the decimal point of the receiving item.



The extra positions are filled with zeros. This is the final result:



●●●

56 If the source value is 95v4 and the receiving picture is 999V99, the value of the receiving item following a numeric move will be _____.

●●●

095v40

57 Assume source value is 6v5 and receiving picture is 9V999 in a numeric move. The result will be _____.

●●●

6v500

58 Take one more numeric move. Source value: 25. Receiving picture: 9999V99. Result: _____.

●●●

0025v00

59 After this MOVE statement is executed, the value of QUANTITY will be _____. (The picture of QUANTITY is 99V999.)

```
    MOVE 1 TO QUANTITY.
```

●●●

01v000

60 When there are different numbers of decimal places in the source and receiving items, as in the above example, the compiler will instruct the computer to shift the data over the appropriate number of positions. Naturally, the object program will be more efficient if we can do without such shift instructions; we can avoid them by making sure that the source and receiving items have the same number of decimal places.

On a COBOL program sheet, rewrite the MOVE statement shown in the preceding frame so that it will not be necessary for the computer to shift the source value during the move.

●●●

```
    MOVE 1.000 TO QUANTITY.
```

61 Numeric move. When the decimal points are aligned...

Rule 5. Truncation. Any digits for which there is no room in the receiving item are truncated. Truncation can occur at either or both ends.

This rule means that you can lose digits during a move -- and they can be very significant digits. Consider the MOVE statement below.

```
    MOVE 1000.00 TO AMOUNT.
```

If a picture of AMOUNT is S999V99, the value of AMOUNT after the move will be _____.

●●●

000v00

62 Although losing significant digits is probably not what you had in mind, the computer will do exactly what your COBOL program has indicated. Every now and then, to be sure, you may want to drop certain parts of an item. Imagine that you have an item whose picture is S9999V999.

(1) What picture would you give to a receiving item if you wanted to move only the whole number and not the fraction? (These might be dollars-and-cents, and you want to move only the whole dollars.)

(2) What picture would you give to the receiving item if you wanted to move only the fraction?

(3) What picture would you give to the receiving item if you wanted to move the entire number, but with only two decimal places instead of three?

●●●

(1) S9999
(2) SV999
(3) S9999V99

63 *Let's carry this example a step further. Suppose that what you want to move is just one digit, the leftmost digit, which represents the number of thousands. Thus, if the value of our item is 7263v05, all you want to move is the digit 7.*

It should be clear that if you made the picture of the receiving item just 9, the digit you would move is the units digit: 3. It is necessary to introduce three "assumed positions" into the picture in order to obtain the desired alignment. The correct picture for this problem would be 9PPP (or 9PPPV). This picture describes a one-digit item, you will recall, because no storage space is reserved for Ps in a picture.

Assume that a source value is 025936v45, and you wish to move only the thousands (025). The receiving item picture must be _____.

●●●

999PPPV

64 Now that you have seen the effect of having Ps in the receiving picture, consider how Ps affect a move when they are in the source picture. There, they serve as "assumed zeros".

Work these exercises out on scratch paper:

(1) The source value is 256pppv, and the receiving picture is 9(7)V99. After the move, the receiving item will contain _____.

(2) The source value is vpppp797, and the receiving picture is 9V9(6). After the move, the receiving item will contain _____.

●●●

(1) 0256000v00
(2) 0v000079

65 *You have studied all of the rules that govern actions during a numeric move. Before we go on to the edit move, though, I want to pick up a loose thread from way back in Lesson 1. When we were discussing the fact that some moves are valid while others are not, we looked at a sample MOVE statement (MOVE DATE TO INVOICE-DATE) and decided that the move was not valid, even though both items contained numeric data. The reason for this was that DATE happened to be a group item, while INVOICE-DATE was an elementary item.*

Now in fact, these data items can be redefined in such a way as to make a valid move possible! For instance, it is possible to redefine the group item (DATE) as an elementary numeric item. And, for that matter, the number item (INVOICE-DATE) can be redefined as an alphanumeric item. Depending on how we define the items, we would cause either a numeric move or an alphanumeric move!

This idea goes well beyond the bounds of this little problem. The key point is that you, the programmer, define the data items, and depending on how you define them, you may make moves possible or impossible. Also, you determine whether an alphanumeric or a numeric move takes place. In general, you define an item once, and you decide, for instance, whether to make the picture of a number 9(5) or X(5). But the ability to redefine items is a powerful tool in your bag of COBOL programming techniques, because it permits you to have two or more definitions for the same item.

Reading assignment: REDEFINES clause

Also, re-read the summary of the REDEFINES clause in the COBOL Program Fundamentals reference handbook (under Item Description Entries in the Data Division section).

●●●

66 These were the Data division entries you read in Lesson 1 when we decided that it was not legal to MOVE DATE TO INVOICE-DATE:

```
    03  DATE.
        04  MONTH, PICTURE 99.
        04  DAY, PICTURE 99.
        04  YEAR, PICTURE 99.

02  INVOICE-DATE, PICTURE 9(6).
```

Take a COBOL program sheet, and on it write an entry that redefines DATE as an elementary numeric item named CURRENT-DATE.

•••

```
    03  CURRENT-DATE, REDEFINES DATE,
        PICTURE 9(6).
```

67 The entry you have just written would appear

{ on the line above 03 DATE.
on the line below 03 DATE.
on the line below 04 YEAR, PICTURE 99. }

•••

on the line below 04 YEAR, PICTURE 99.

68 Does the entry you wrote make it valid to MOVE DATE TO INVOICE-DATE?

•••

No, since DATE is still a group item. However, it does make it valid to MOVE CURRENT-DATE TO INVOICE-DATE.

69 If we wanted to cause an alphanumeric instead of a numeric move, we would redefine INVOICE-DATE as an alphanumeric item. On your program sheet, write this redefinition. Name the new item R-INVOICE-DATE.

•••

```
02  R-INVOICE-DATE, REDEFINES
    INVOICE-DATE, PICTURE X(6).
```

70 Two valid MOVE statements can be written to move the date into R-INVOICE-DATE. On the program sheet, write both of those statements.

●●●

```
    MOVE DATE TO R-INVOICE-DATE.

    MOVE CURRENT-DATE TO R-INVOICE-DATE.
```

71 *All of the redefining that we have been doing has been predicated on the notion that the original definitions were necessary for some other processing being done with those items. Otherwise, it would not have made sense to go to the trouble of redefining the items, and we could have just changed the picture of INVOICE-DATE from 9(6) to X(6), to make MOVE DATE TO INVOICE-DATE a valid move.*

The point is that redefinition makes sense only if two or more definitions of an item are really necessary, for instance, when an item will be processed in two or more different ways. Suppose that we have an item called ITEM-A; at one point in a procedure, a number must be moved from an alphanumeric item to ITEM-A; later in the procedure, the number must be moved from ITEM-A to an elementary numeric item.

ITEM-A seems to be caught in the middle. If we define ITEM-A as an alphanumeric item, the second move will not be valid. If we define ITEM-A as a numeric item, the first move will not be valid. Based on what we know about this problem, it certainly appears that ITEM-A must be both alphanumeric and numeric! This is a legitimate place to employ redefinition.

On a program sheet, define this item and then redefine it. First, define it as a 5-character alphanumeric item named ITEM-A1; then, redefine it as ITEM-A2, a 5-digit numeric item. Both times, the level number is 02. (As you have read, you cannot change the level number when you redefine an item.)

●●●

```
    02  ITEM-A1, PICTURE X(5).
    02  ITEM-A2, REDEFINES ITEM-A1,
        PICTURE 9(5).
```

LESSON 3

72 *As we get into edit moves, you will discover that they resemble numeric moves in many ways. (These two types of moves, as you know, are treated as one type in the reference manual.) There are basic similarities when it comes to alignment of data by decimal points, truncation, zero fill, and the treatment of signs.*

But the pictures of the receiving items (report items) are very different, and can become extremely complicated. It is these pictures that we will spend most of our time on during this lesson. Instead of dealing with the rules that govern edit moves, you will learn the rules for causing certain kinds of editing to occur -- that is, you will learn to write report pictures.

On the whole, you will work with pictures in two ways. First, given the picture of the receiving item, and the source value, you will be asked to determine what the result of the move would be. Second, given the source value and the desired result, you will formulate the necessary picture for the receiving item.

●●●

73 Alphanumeric and alphabetic data cannot be edited using the edit move. An edit move occurs only when an elementary _____ item is moved to an elementary _____ item.

●●●

numeric; report

74 Numeric data is edited when it is moved { to / from } a report item.

●●●

to

75 "Editing" means [deleting] [replacing] [inserting] characters in an item.

●●●

ALL of these may be involved in editing, although they may not be all required in every edit move.

76 Here is an example of an edit move. When the source value 00762401 is moved to an item whose picture is $ZZZ,ZZZ.99, the result is$bb7,624.01. (Remember, the little "b's" represent blank spaces in an item's value; nothing would print where the b's are shown.)

Compare the source value with the result. What characters are deleted? What characters were inserted?

●●●

The high-order (leading) zeros were deleted. A dollar sign, comma, and decimal point were inserted.

77 *Here is how editing is actually done:*

1. The compiler uses the report picture you write in a program to make the "pattern" of characters which the System/360 needs for editing. The pattern is stored as a constant in the object program. The compiler also reserves a work area of storage in which the editing will take place.

2. During the execution of the move, first the pattern is moved to the work area. Then the data from the source item is moved to the work area and edited by the computer. Finally, the edited result is moved from the work area to the receiving item.

This explanation may seem brief, but it is the whole story, short of getting into the technicalities of internal operations of the System/360 -- about which, as COBOL programmers, we frankly couldn't care less.

●●●

78 Here are three questions asked by COBOL students. See if you can answer them, based on the explanation given in the preceding frame.

(1) Is a report picture put into the object program?
(2) If the picture is stored in the receiving item, won't it be destroyed as soon as some data is moved into it?
(3) Is the editing done in the receiving item?

●●●

The answer to all three questions is NO. (1) The COBOL report picture is converted into a System/360 edit pattern. (2) The picture itself is not around at all during the execution of the object program; its equivalent, the edit pattern, is stored elsewhere with other constants until it is needed, and is not stored in the receiving item. (3) Editing is done in a work area, not in the receiving item; only the edited result is moved to the receiving item.

79 *We see that the way in which the object program edits the data is based on the picture we write for the report item. Each editing picture character causes one or more actions to take place during editing. Therefore, literally thousands of combinations of actions are possible. While we will study all of the picture characters, don't expect us to study all of the possible combinations of characters!*

Reading assignment: PICTURE clause, Report-form option

Important: This reading assignment contains a great deal of detailed information. Don't try to absorb it all at once! Instead, merely scan the material at this time, to get an idea of the kinds of characters that can appear in report pictures. As we study the characters in detail -- and we will do this one, two, or three characters at a time -- go back to the reference manual to re-read the information about those characters carefully.

●●●

80 Edit rule 1. To move digits just as in a numeric move, use picture characters "9", "V", and "P" the same as in pictures of numeric items. "S" cannot be used.

All five of the actions we discussed for numeric moves can occur in the edit move:

1. Conversion. Remember that the usage of report items is always DISPLAY. Therefore, we will always wind up with the edited result in BCD (external decimal), regardless of the usage of the source item.

2. No sign. There cannot be an S in a report picture, so "no sign" bits will be developed in the edited result.

3. Alignment by decimal points.

4. Zero fill in any extra positions at either or both ends.

5. Truncation of digits for which there is no room, at either or both ends.

●●●

81 In report pictures, the character "9" defines a position into which a digit is to be placed without being changed in any way. As an illustration, the picture 99,999 calls for the insertion of a comma into a number, but indicates that the digits themselves are not to be changed in any way. So, if the source value were 00065, the edited result would be _____.

●●●

00,065

82 We say that "9" defines a digit position in the picture. Later you will learn that the characters "Z", "*", and in certain cases, "$", "+", and "-" also define digit positions.

By contrast, "P" and "V" do not define positions into which digits are put. As you already know, these characters are used to _____.

●●●

align the data

83 Edit rule 2. To insert an actual decimal point, write "." in the picture. This takes the place of an assumed decimal point in aligning the data. A picture cannot contain both a "." and a "V".

●●●

84 The character "." is a real character which actually occupies a byte of storage in the edited result. Whereas the picture 999V99 defines an item that is _____ bytes long, the picture 999.99 defines an item that is _____ bytes long.

●●●

5; 6

85 A report item's picture is 999.99; what will the edited results be if the source value is (1) 123v45, (2) 123, (3) 1v23, and (4) 1234v567?

●●●

(1) 123.45 (2) 123.00 (3) 001.23 (4) 234.56

86 Edit rule 3. To suppress leading zeros and replace them with blanks or asterisks, write "Z" or "*" in each digit position in which you want suppression. A picture cannot contain both an "*" and a "Z".

●●●

87 In most cases, the zero suppression stops when you get to the end of the Zs or when the first significant digit is reached. Suppose that ZZZZ is the picture of a receiving item; what will be the edited result is the source value is (1) 0500, (2) 0002, (3) 0000?

●●●

(1) b500 (2) bbb2 (3) bbbb

88 Zs and 9s (or *s and 9s) can both appear in a picture, but a Z or an * cannot appear anywhere to the right of a 9. This means that you cannot get zero suppression to start and stop repeatedly, or to occur in the middle of an item; zero suppression is strictly for high-order zeros.

On scratch paper, write a picture that defines five digit positions, with zero suppression in the two leftmost positions.

●●●

ZZ999

89 Write a report picture which will produce the result bbbb.23 from a source value 0000v23.

●●●

ZZZZ.99

90 What report picture will produce the result *****.09 from 00000v09?

●●●

*****.99

Asterisks used in this way are often called "check protection" symbols. They are printed on checks to make it difficult to increase the amount by typing additional digits.

91 Some special rules apply to pictures that call for zero suppression and also contain a decimal point. First of all, you can write Zs or *s to the right of the decimal point; however, zero suppression stops at the decimal point unless the source value is zero.

On scratch paper, write down what the edited result would be in each of these cases:

	Report Picture	Source Value	Edited Result
(1)	ZZZ.ZZ	000v03	?
(2)	ZZZ.99	000v03	?
(3)	ZZVZZZ	00v001	?
(4)	ZZ9.9	00v5	?

●●●

(1) bbb.03 (2) bbb.03 (3) bbv001 (4) bb0.5

92 Here is a second special rule that involves decimal points: Only one kind of digit position character is allowed to appear to the right of the decimal point. For example, a Z may appear to the right of a decimal point only if all digit positions are represented by Zs.

Decide which of these pictures are right and which are wrong:

(1) ZZ.9999
(2) ZZZ.Z99
(3) ****.**
(4) 9999.ZZ

●●●

Pictures (1) and (3) are correct. Picture (2) is wrong because it violates the rule stated in this frame. Picture (4) violates the earlier rule that Zs or *s may not appear anywhere to the right of a 9.

93 This is the last special rule about decimal points: If zero suppression is called for in all digit positions, and the source value is zero, all of the zeros are suppressed and the decimal point is suppressed too.

This rule means that when the source value is zero, the picture ZZZ.ZZ will produce an edited result of

{ bbb.bb
{ bbb.00
{ bbbbbb

●●●

bbbbbb

94 Take careful note of that last special rule. There is an important difference in the end result depending on whether or not the source value is zero. With a receiving picture of Z(5).ZZ, and a source value of 00v00, the edited result is _____. With the same receiving picture, and a source value of 00v05, the result is _____.

●●●

bbbbbbbb; bbbbb.05

95 The same action applies to pictures that have *s in all digit positions, except of course, that all of the suppressed characters -- including the suppressed decimal point -- are replaced by asterisks. If the receiving item's picture is *****.**, and the source value is zero, the result will be _____.

●●●

****

96 Edit Rule 4. To insert commas, spaces, or zeros, write "," where you want a comma, "B" where you want a blank space, and "0" where you want a zero.

●●●

97 Apply this rule in the case of a social security number. The source value is a string of nine digits, such as 300926354. You want to insert blank spaces after the third and fifth digits, to get 300b92b6354 as the edited result. The picture needed to accomplish this editing is _____.

●●●

999B99B9999

98 Commas are the insertion characters you will undoubtedly use most often. Write a picture which will insert commas into the source value 9285406 to produce 9,285,406 as the result.

●●●

9,999,999

99 Write a picture that will do the following things: (a) edit a seven-digit number; (b) insert a decimal point between the dollars and the cents; (c) insert a comma between thousands and hundreds of dollars; (d) suppress zeros up to the decimal point, and replace them with blanks. For example, given the source value 02307v95, the edited result should be b2,307.95.

●●●

ZZ,ZZZ.99

100 Sometimes inserted commas and zeros are not significant characters. If the report picture calls for zero suppression, the inserted characters are suppressed and replaced along with the leading zeros of the source value. For example, if the picture is *,*** and the source value is 0009, the result will be ****9 (not *,**9).

For each source value listed below, figure out the edited result if ZZZ,ZZZ.ZZ is the picture of the receiving item.

	Source Value	Edited Result
(1)	002800v03	?
(2)	000529v61	?
(3)	000000v07	?
(4)	000000v00	?

●●●

(1) bb2,800.03
(2) bbbb529.61
(3) bbbbbbb.07
(4) bbbbbbbbbb

101 Keep in mind that the inserted commas are suppressed only if zero suppression is called for by the report picture. Assume that 0062v25 is the source value; how will the result differ if the report picture is 9,999.99 as opposed to Z,ZZ9.99?

●●●

Result with first picture: 0,062.25
Result with second picture: bbb62.25

102 Edit rule 5. To put a fixed dollar sign to the left of an amount, write "$" where the dollar sign is to appear.

●●●

103 When we speak of a "fixed" dollar sign, we mean one that stays in one position. (Later you will study about "floating" dollar signs.)

On scratch paper, write the edited results in these cases:

	Report Picture	Source Value	Edited Result
(1)	$Z,ZZZ.99	3001v02	?
(2)	$Z,ZZZ.99	0982v75	?
(3)	$Z,ZZZ.ZZ	0000v04	?

●●●

(1) $3,001.02 (2) $bb982.75 (3) $bbbbb.04

104 Take the case where all digit positions in a picture are represented by Zs or *s. You have learned that in such a case, when the source value is zero, not only the zeros but also the commas and decimal point are suppressed. It would be pretty silly to print a fixed dollar sign in this situation, so the dollar sign is also suppressed.

What will the edited results be if the source value is zero and the report picture is (1) $ZZZ or (2) $Z,ZZZ,ZZ?

●●●

(1) bbbb (2) bbbbbbbbb

105 If the report picture is $***.** and the source value is zero, the edited result will be a string of

{ 5 asterisks
{ 6 asterisks
{ 7 asterisks

●●●

7 asterisks (Both the dollar sign and the decimal point are replaced by *s.)

106 Observe the difference in edited results when the source value is zero and the report picture is (1) $Z,ZZZ.99 and (2) $Z,ZZZ.ZZ. Write down the two results on scratch paper.

●●●

(1) $bbbbb.00 (2) bbbbbbbbb

107 *Incidentally, another way of achieving a result of all-blanks when the source value is zero is to write a BLANK WHEN ZERO clause in the description of the report item.*

Reading assignment: BLANK WHEN ZERO clause. (Also referred to in the manual simply as the BLANK clause.)

●●●

108 *These two ways of defining a report item would produce identical edited results:*

```
    02  INVENTORY-WORTH, PICTURE $Z,ZZZ.ZZ.
```

```
    02  INVENTORY-WORTH, PICTURE $Z,ZZZ.99,
        BLANK WHEN ZERO.
```

The BLANK WHEN ZERO method might give slightly more efficient action when the source value is likely to be zero fairly frequently. This clause causes the value of the source item to be tested before any editing is done; if the value is zero, no editing is done, and the receiving item is just blanked out. The important thing, though, is that the end result is the same either way you do it.

●●●

109 Edit rule 6. To identify negative values with CR, DB, or - (but to have no identification for positive values), write "CR" or "DB" at the right end of the picture, or write "-" as either the first or last character of the picture.

To identify a value with + when it is positive and - when it is negative, write "+" as either the first or last character of the picture.

●●●

110 *The sign indicators covered by this rule are "fixed" indicators. That is, the characters remain in the position you specify. (Later, we will discuss "floating" plus and minus signs.)*

Only one kind of sign indication can be used in a picture: CR or DB or - or +. Which of these is the only one that can be used to identify positive values?

●●●

+

111 CR, DB, and - all identify negative values. When one of these is used in a picture, and the source value is positive, what sign identification appears in the edited result?

●●●

None. Blanks appear where the sign indicator would have been for a negative value. (One blank in the case of the minus sign; two blanks for CR or DB, since each of these occupies two character positions.)

112 If + is used in a picture, and the source value is negative, what sign identification appears in the edited result?

●●●

-

113 Let's compare the results when + and - are used in pictures. Jot down the edited result on scratch paper.

	Source value	Edited result when report picture is +999	Edited result when report picture is -999
(1)	005	?	?
(2)	-613	?	?
(3)	000	?	?

●●●

When picture is +999: (1) +005 (2) -613 (3) b000
When picture is -999: (1) b005 (2) -613 (3) b000

In COBOL, you will recall, zero is neither positive nor negative, so there is never any sign indication for a zero value.

114 What will be the edited results if the report picture is ZZZZ- and the source value is (1) -0265 and (2) 0053?

●●●

(1) b265- (2) bb53b

115 Write a single report picture that will produce these edited results (from different source values):

$1,005.67bbb
$bbb35.00bCR
$bbbbb.50bbb

(Notice that a blank has been inserted between the amount and the sign indication.)

●●●

$Z,ZZZ.99BCR (or $Z,ZZZ.ZZBCR)

116 Edit rule 7. To "float" a dollar sign, plus sign, or minus sign up to the first significant digit, and at the same time, to replace leading non-significant characters with blanks:

a. Write a string of "$" signs to get a floating dollar sign.
b. Write a string of "+" signs to get a floating + sign when the value is positive, and a floating - sign when the value is negative.
c. Write a string of "-" signs to get a floating - sign when the value is negative, but no sign indicator when the value is positive.

A picture can contain only one of these "floating strings".

●●●

117 *To begin with, we will talk about floating dollar signs only; however, everything we will say also applies to floating plus signs and minus signs. Let's take a moment to look at two columns of figures, just to see the difference between amounts printed with a fixed dollar sign and those printed with a floating dollar sign.*

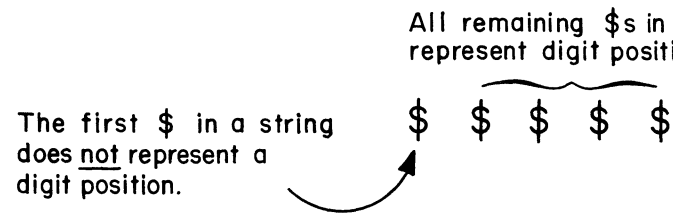
$25,000.00	$25,000.00
$ 250.00	$250.00
$ .25	$.25

●●●

118 To get a fixed dollar sign and zero suppression for a four-digit number, we can write this report picture: $ZZZZ.

To get a floating dollar sign and zero suppression for the same number, we write this report picture: $$$$$.

Not all of the dollar signs in the "floating string" have the same significance, as illustrated below.



Thus, the rule for a floating string is that the string must contain one extra $ in addition to a $ for each digit position. This means that a report item whose picture is $$$$$$ is large enough to hold up to _____ digits.

●●●

five

119 The largest amount that can be edited with the picture $$$ is _____.

●●●

99

120 Insertion characters (, B 0) can be put into a floating string. The dollar sign "floats through" the insertion characters to get right next to the first significant digit. For instance, a report picture might be $$,$$$; if the value moved to this item were 0362, the edited result would be _____.

●●●

bb$362 (The dollar sign has floated through to replace the comma.)

121 A picture with a $-string doesn't have to be composed entirely of dollar signs. You can end the $-string at the rightmost position to which you want the $ to float. The remaining digit positions must then be represented by 9s.

For instance, the picture $$,$$$.99 allows the $ to float up to the decimal point only. With this picture, a source value of 0000v05 would be edited like this: bbbbb$.05

How would you change this picture in order to make the result bbbb$0.05?

●●●

$$,$$9.99

122 The preceding frame pointed out that any digit positions to the right of a floating string must be represented by 9s. Zs and *s, then, are not allowed. But let's say that you are editing a dollars-and-cents amount with a floating dollar sign, and you want the edited result to be all blanks when the amount is zero. You must not write $$$$.ZZ, but there are two other things that you can do.

One is to write $$$$.99 as the report picture, and specify BLANK WHEN ZERO. The second is a special extension of the floating string designed just for this purpose: write dollar signs in all positions to the right of the decimal point -- $$$$.$$.

In this special picture, with all digit positions represented by dollar signs, the $ never actually floats to the right of the decimal point. If the source value is less than one dollar, the $ floats up to the decimal point. If the source value is zero, the edited result is all blanks.

To see if you understand this action, figure out what the edited results will be in the four cases below. In each case, $$$,$$$.$$ is the picture of the receiving item.

	Source Value	Edited Result
(1)	02000v00	?
(2)	00030v00	?
(3)	00000v04	?
(4)	00000v00	?

●●●

(1) b$2,000.00 (2) bbbb$30.00 (3) bbbbbb$.04 (4) bbbbbbbbbb

123 Floating strings of plus or minus signs are like strings of dollar signs in all respects.

Suppose that the picture of a source item is S9(4); write the picture of a receiving item which will identify the sign of the item with a floating + if the value is positive and a floating - if the value is negative.

●●●

++,+++ or +++++ or +(5)

124 What is wrong with this picture: -,---.99?

●●●

The picture is correct in every respect, except that it doesn't make sense to have only one minus sign to the left of the comma. The first symbol in a floating string is an extra symbol which does not represent a digit position. As it stands, the report item will accommodate hundreds of dollars, and might better have been written as ----.99; however, if the item was intended to hold thousands of dollars, then --,---.99 would be the correct picture.

125 Suppose that --- is the picture of a receiving item. What will be the edited results when the source values below are moved to the item?

	Source Value	Edited Result
(1)	02	?
(2)	00	?
(3)	-50	?
(4)	-105	?

●●●

(1) bb2 (2) bbb (3) -50 (4) b-5

The fourth source value is a little sneaky. I included it to remind you that truncation occurs when the source value is longer than the receiving item. This receiving item has positions for two digits, and the assumed decimal point is at its right end. Only 05 is moved to the item; then the zero is suppressed, and the - is floated up to the 5.

126 *You have now learned about all of the editing picture characters, but it would not be fair to leave the subject of editing without at least a brief discussion of two topics: first, how to edit alphabetic or alphanumeric data, and second, how to edit numeric data when no permissible report picture suits your needs.*

Actually, these two topics are just two aspects of one programming technique. The technique is to "assemble" the desired result by using a series of moves.

●●●

127 A typical instance in which such assembling must be done is when a name that looks like this in an input record: "DRUNKENBURGERbbbbbbbCS", must look like this on a printed report: "C. S. DRUNKENBURGERbbbbbbb". To handle this, we define an area in working storage, with spaces and periods in the appropriate positions, and with room for the initials and last name. Three moves are then needed to put the initials and last name where we want them, and a fourth move transfers the edited name to the output record.

Let's take it step by step. This is how the name has been defined in the input record:

```
    02  NAME.
        03  LAST-NAME, PICTURE A(20).
        03  FIRST-INITIAL, PICTURE A.
        03  SECOND-INITIAL, PICTURE A.
```

On a program sheet, write the entries to define a working-storage record called EDITED-NAME, with INITIAL-1 in the first position, followed by a period and a space; then INITIAL-2 in the fourth position, another period and space; finally 20 postions for SURNAME.

●●●

```
WORKING-STORAGE SECTION.

01  EDITED-NAME.
    02  INITIAL-1, PICTURE A.
    02  FILLER, PICTURE XX, VALUE '. '.
    02  INITIAL-2, PICTURE A.
    02  FILLER, PICTURE XX, VALUE '. '.
    02  SURNAME, PICTURE A(20).
```

128 Now write the MOVE statements to put the pieces of the name into their proper places in the working-storage record. (To avoid complicating the problem, let's assume that everybody has two initials.)

Finally, write a MOVE statement to transfer the edited name to CUSTOMER-NAME. (Assume that CUSTOMER-NAME is a 26-position alphanumeric item in the output record, which has already been defined.)

●●●

```
MOVE FIRST-INITIAL TO INITIAL-1.
MOVE SECOND-INITIAL TO INITIAL-2.
MOVE LAST-NAME TO SURNAME.
MOVE EDITED-NAME TO CUSTOMER-NAME.
```

129 Now that you have the hang of it, try applying the technique to editing numeric data in a way which no single report picture will allow. Specifically, the problem is to edit a date in this way: b4/b5/66 (when the source value is 040566).

A side comment: Beginners sometimes get the impression that they can do absolutely anything with report pictures, and are tempted to invent new pictures -- such as Z9/Z9/99 for this problem. This is not a legal picture, and the compiler will reject it. Your best bet is to strictly follow the rules you have studied -- the rules given in the reference manual. Take the attitude that if the manual doesn't say you can do it, you can't do it!

For this problem, our approach will be to define a working-storage record called TODAYS-DATE, composed of two positions called MO, then one position containing a /, two more positions called DA, another /, and then two final positions called YR. The pictures of MO and DA must both call for zero suppression in their left positions. Write the entries for this record on a program sheet.

●●●

```
WORKING-STORAGE SECTION.

01  TODAYS-DATE.
    02  MO, PICTURE Z9,
    02  FILLER, PICTURE X, VALUE '/'.
    02  DA, PICTURE Z9.
    02  FILLER, PICTURE X, VALUE '/'.
    02  YR, PICTURE 99.
```

130 This is the data which we are to edit by moving it into our working-storage area. On the coding sheet, write the necessary MOVE statements.

```
    02  DATE.
        03  MONTH, PICTURE 99.
        03  DAY, PICTURE 99.
        03  YEAR, PICTURE 99.
```

●●●

```
    MOVE MONTH TO MO.
    MOVE DAY TO DA.
    MOVE YEAR TO YR.
```

The same basic technique could have been used to edit the date with hyphens (for example, b4-b5-66).

131 *From these little problems, you can see that it is more cumbersome to edit information as soon as you get out of the range of the "built-in" editing capabilities of the language. It is certainly easier to write a single MOVE statement to a report item, and get the editing done that way!*

This suggests that it might be wise to re-evaluate any editing requirements that fall outside the range of built-in language capabilities. In the case of the date-editing problem you just worked on, we might ask, is it necessary for this report to have the date edited like "4/ 5/66" or "4 - 5-66"? Or, would "04 05 66" serve our purposes just as well? If the latter format will do, we can insert the blanks in one simple edit move, by making 99B99B99 the picture of the receiving item.

●●●

LESSON 4

132 *In this lesson, you will study arithmetic operations in detail. Although "arithmetic" may seem like it should be entirely different from "data moving and editing", you will discover that many of the actions that go on are the same. In fact, one of the main steps in executing an arithmetic statement consists of making either a numeric move or an edit move! So don't purge your mind of what you have learned about moves in the first three lessons; also, don't hesitate to go back over the material you have previously studied -- either in this textbook or in the reference manual -- if you find that you have forgotten some of it.*

Reading assignment: Arithmetic statements

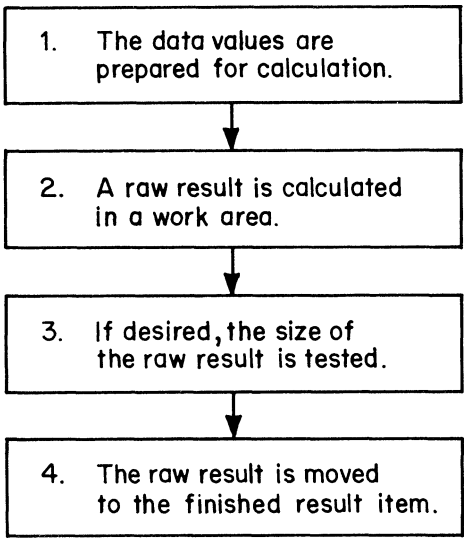
Read through the entire section on arithmetic in the manual once, briefly. Later in this lesson, you will be instructed to re-read the information about certain clauses with greater care and concentration.

●●●

133 *This book deals only with arithmetic done on items whose usage is DISPLAY, COMPUTATIONAL, or COMPUTATIONAL-3. In other words, you will not be taught the details of "floating-point" arithmetic, which involves COMPUTATIONAL-1 and COMPUTATIONAL-2 items. This topic was omitted mainly because comparatively few COBOL users employ floating-point items, and also because it would tend to take us away from our main subject. You see, the question of whether or not to use floating-point items is mainly a system design question, and doesn't affect the way arithmetic statements are written in COBOL.*

●●●

134 *Four major actions occur during the execution of an arithmetic statement. This chart shows what the four actions are and the sequence in which they occur. During this lesson, you will study these actions one at a time.*



●●●

135 Arithmetic action 1. The data values are prepared for calculation. If necessary, the usage of the values is converted to a data code in which calculation is possible. Also, the sign bits are changed in certain cases.

●●●

136 Arithmetic operations can be performed with certain kinds of items, and with no other kinds. Specifically, computations may be done using [elementary numeric items] [elementary alphanumeric items] [report items] [group items] [numeric literals] [non-numeric literals].

●●●

ONLY elementary numeric items and numeric literals

137 To this COBOL restriction, we must add a fact about the System/360. The computer can execute arithmetic operations either on binary items or on packed-decimal items. Therefore, in order to execute an arithmetic statement, it is definitely necessary to convert the data code of elementary numeric items whose usage is _____.

●●●

DISPLAY (that is, BCD or external decimal)

138 The computer can perform computations on two packed-decimal items or on two binary items. Therefore, the preparation of an external-decimal (display) item is not the only time conversion is necessary. Another time would be when one item is packed decimal and the other is binary; one of these items must be converted to the code of the other before computation can begin.

The only case in which <u>both</u> items must be converted is when

{ both are binary.
one is external decimal, and the other is binary.
one is external decimal, and the other is packed decimal.
both are external decimal.
both are packed decimal.
one is packed decimal, and the other is binary. }

●●●

both are external decimal

(Both items will be converted to packed-decimal format.)

139 Imagine that this statement appears in a COBOL program:

```
ADD CHECK-AMOUNT TO FLOAT-TOTAL.
```

Here are the descriptions of the items to be added:

```
02  CHECK-AMOUNT, PICTURE S9(6)V99,
    COMPUTATIONAL-3.

77  FLOAT-TOTAL, PICTURE 9(8)V99.
```

(1) Is it permissible to add these items? Why?
(2) Will it be necessary to convert the data code of either item? If so, which item, and to what data code must it be converted?

●●●

(1) The items can be added because both are elementary numeric items.
(2) FLOAT-TOTAL must be converted since it is a display (BCD) item; it will be converted to packed decimal because that is the data code of CHECK-AMOUNT.

140 Although we typically say that an "item" is converted, it would be more accurate to say that the data from an item is converted to a different code. It works like this: the data is moved from the item to a work area, and converted there; then, the arithmetic operation is done using the data in the work area. The contents of the original data item remain unchanged.

The work area, by the way, is set up by the compiler, not by the programmer. And the compiler also generates all of the necessary instructions to cause conversion.

Note carefully that the converted data is used only for one calculation. The next data conversion that is required will probably be done in the same work area, erasing the previously-converted data.

Check your understanding of these ideas by applying them to this example.

```
SUBTRACT STOCK-COUNT FROM
    EXPECTED-BALANCE GIVING LOSS.
```

If STOCK-COUNT is an external-decimal item, and EXPECTED-BALANCE is a packed-decimal item, the data from STOCK-COUNT is moved to a work area in which it is converted to packed-decimal. Then subtraction is done, using the data from

{ STOCK-COUNT / the work area }.

●●●

the work area

141 Suppose that STOCK-COUNT is involved in two later calculations in other procedures of the same program. What will happen when the later calculations are performed?

(The data from STOCK-COUNT will be converted again each time it is used in a calculation. / The converted data from STOCK-COUNT will be saved in the work area for use in the later calculations. / The compiler will generate instructions to bypass the later calculations, to avoid using STOCK-COUNT in more than one computation.)

●●●

The data from STOCK-COUNT will be converted again each time it is used in a calculation.

142 We generally want to prevent repeated conversions. We can do this by making certain that the items we use in calculations have the proper usage. In the example given in the preceding frames, repeated conversions would not have been necessary if the usage of the "stock count" data had been COMPUTATIONAL-3 (packed decimal).

There are two ways that this can be done. First, by redesigning the input record to have the data in packed-decimal format when it enters storage for processing. This makes it unnecessary to convert the data when it is used in calculations; however, it may be necessary to convert the data to DISPLAY format elsewhere in the program, if the data is printed or punched. The system designer or programmer has to evaluate the overall processing requirements to figure out whether the data will more often be used in calculations, in which case COMPUTATIONAL-3 usage is better; or more often printed, in which case DISPLAY usage is the right choice.

Whereas the first way of avoiding conversion is a system design problem, the second way is a programming technique. This way is to define an item in working storage whose usage is COMPUTATIONAL-3, to MOVE the data to that item (thereby causing it to be converted), and thereafter to use the working-storage item in calculations of the data. In this way, the data is converted only once, no matter how many times it is used in calculations.

This programming technique is an efficient way of treating an external-decimal item,

{ provided that the item is used in one calculation only.
if the item is used in two or more calculations.
no matter how many calculations the item is used in. }

●●●

if the item is used in two or more calculations

143 In the STOCK-COUNT example, we might previously have written the statement "MOVE STOCK-COUNT TO CALC-STOCK-COUNT", with CALC-STOCK-COUNT defined as a COMPUTATIONAL-3 item. To calculate using this data, we would later have written "SUBTRACT _____ FROM EXPECTED-BALANCE GIVING LOSS".

●●●

CALC-STOCK-~~BALANCE~~ COUNT

144 Based on the picture of an item that is to be used in a calculation, the compiler may also generate instructions to alter its sign. Signs are always considered by the computer in arithmetic operations; part of the computer's job, of course, is to figure out the correct sign of a result.

The sign of a data value will be altered if there is no S in the item's picture. Special "no sign" bits will replace the operational sign bits in this case. The "no sign" bits are interpreted as positive during calculations.

One reason you might omit the S from an item's picture would be to use the "absolute value" of the item in calculations. The absolute value of an item is its numerical value disregarding the sign. Suppose an item's value is -826.036, but we want to treat it as 826.036 in a calculation; we accomplish this by making the picture of the item

```
{ 9(6)       }
{ S999V999   }
{ 999V999    }
{ NO-S999V999}
```

●●●

999V999

You will recall that S can be omitted from pictures of DISPLAY and COMPUTATIONAL-3 items, but not COMPUTATIONAL items.

145 Take the opposite situation: you know that an item has no sign, so you want to avoid the unnecessary insertion of "no sign" bits. The picture you might write for such an item is

```
{ S9(5) }
{ 9(5)  }
{ 99999 }
```

●●●

S9(5)

By putting an S in the picture, we in effect say to the compiler, "Use the sign bits -- or no-sign bits -- which this item already contains." By omitting the S, we in effect say, "Change the sign bits of this item's value to no-sign".

146 Arithmetic action 2. A raw result is calculated in a work area. At this step, the actual adding, subtracting, multiplying, dividing, or exponentiating is done. The "raw" result is the numerical outcome of the calculation, such as 000306v4294. (The "finished" result in this case might be $306.43.)

The compiler sets up the work areas needed for calculations. (You do not define these areas in the Working-Storage section.)

●●●

147 The compiler makes the work area large enough to develop the result you want -- based on the operation (addition, subtraction, multiplication, division, or exponentiation) and the pictures of the items. For example, when two numbers, both having the picture S999V99, are added, the work area need only contain six positions; but if the same numbers are multiplied, a ten-position work area is needed.

Furthermore, the size of the work area is adjusted to take account of any shifting of data values that is needed to align decimal points. If the pictures of three numbers to be added together were 99V99, 9V9999, and 9999V9, the work area would be _____ positions long.

●●●

nine

This may be easier for you to see when values are given to each data item and the numbers are aligned, like this:

```
  25v75
   8v0036
+ 9981v2
  10014v9536
```

In our discussion of numeric moves, I pointed out that alignment by decimal points requires shifting when the numbers do not have the same number of decimal places. The same point applies to arithmetic. If you are concerned about saving storage space and making your programs as efficient as possible, try to have the same number of decimal places in values that are added or subtracted.

148 Arithmetic action 3. If desired, the size of the raw result is tested. A test can be made to determine whether all of the significant integral digits will fit into the finished result item. The test is made only if an ON SIZE ERROR clause follows the arithmetic statement.

If all significant integral digits of the raw result will fit into the finished result item, the fourth arithmetic action is performed -- the raw result is moved to the finished result item. If they will not fit, the raw result is not moved; instead, the execution of the arithmetic statement is suspended, and control goes to the statements that are written in the SIZE ERROR clause.

●●●

149 *I will explain what all this means, but before I do, carefully re-read what the manual says about the SIZE ERROR test.*

Reading assignment: Arithmetic statements, SIZE ERROR option

●●●

150 The integral digits of a number are those that appear in the integral places of the number.

Let's define what we mean by "integral places". Decimal places are the ones to the right of the actual or assumed decimal point. Integral places are the ones to the _____ of the decimal point.

●●●

left

151 An item whose picture is 999V99 has $\left\{\begin{matrix}5\\3\\2\end{matrix}\right\}$ integral places.

●●●

3

152 Suppose the value of an item is 3924v00895. The integral digits are _____.

●●●

3924

153 In the value 000306v2, which integral digits are significant? Which are not significant?

•••

significant
integral digits

000306v2

insignificant
integral digits

154 If the raw result 001002v05 were moved to a finished result item, whose picture is 999V99, would any significant integral digits be lost?

•••

Yes, the 1 in the thousands place would be truncated along with the two insignificant zeros. The finished result after the move would be 002v05!

155 *The SIZE ERROR test determines whether all significant integral digits in the raw result will fit into the finished result items. Notice that only the integral places are tested, not the decimal places. This is because it is normal to drop excess decimal digits; for instance, if an employee's net earnings are calculated to be $175.98270635, he will be paid $175.98. But it generally isn't considered quite fair to drop off digits at the other end of the number!*

It may be wise also for me to spell out what I mean by the "finished result" item. I mean the item in which the result will appear after the execution of an arithmetic statement has been completed.

•••

156 The test is made {before / after} the raw result is moved to the finished result item.

•••

before

157 If an error exists, the result {is / is not} moved.

•••

is not

158 What is the fallacy in this reasoning:

"It doesn't do a bit of good to test for a size error during the execution of a statement such as ADD A TO B. As soon as A and B are added, the original value of B is changed -- and it's too late then to find out that the sum won't fit into B."

●●●

This argument assumes that ADD A TO B is executed by adding the value of A directly into item B. This is not the case, however; A and B are added together in a work area. The raw result is tested in the work area for a size error condition. If the sum is too large to fit into B, it will not be moved into B; therefore, the original values of both A and B remain unchanged.

159 *Return to your reference manual. This time look at the formats of the arithmetic statements to see where the SIZE ERROR clause is written.*

●●●

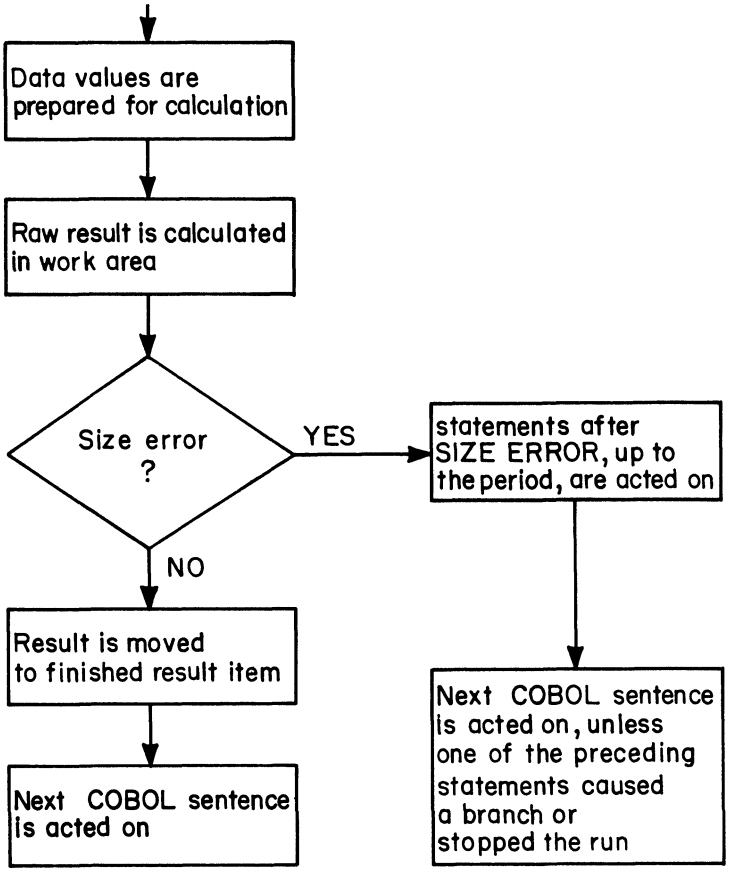
160 Rewrite this statement on a program sheet. Add to it a SIZE ERROR clause which will cause a branch to OVERSIZE-SUM if a size error occurs.

```
ADD OVERTIME TO REGULAR-EARNINGS.
```

●●●

```
ADD OVERTIME TO REGULAR-EARNINGS,
    ON SIZE ERROR, GO TO OVERSIZE-SUM.
```

161 *This chart shows the flow of control through an arithmetic sentence that contains a SIZE ERROR clause.*



Notice that the logic of a SIZE ERROR clause is like that of an IF sentence. In particular, notice the importance of terminating the statements that follow the words "SIZE ERROR" with a period. Thus, an arithmetic "statement" must in fact be a sentence when the SIZE ERROR option is specified.

●●●

162 *It is possible that you will not find much use for size error testing in your programs. After all, the test is not needed if you have allowed enough room in the finished result item; for instance, if two three-digit numbers are to be added, the result cannot possibly exceed four digits -- so if the finished result is put into a four-digit item, there is no need for a size error test. It would be folly to write unnecessary size error tests, of course, since extra instructions are generated in the object program each time a SIZE ERROR clause appears.*

Also, it is unwise to put a lot of SIZE ERROR clauses into a program to catch programming errors in miscalculating the sizes of results. This would be an expensive substitute for thorough testing and debugging of the program!

●●●

163 *Here is a case in which the programmer is probably justified in using the size error test. Four numbers (A, B, C, and D) are to be multiplied to get their product, X. A, B, C, and D each contain six digits; that is, the picture of each one is S9(6). The picture of X is S9(18), which is the largest allowable size for a numeric item.*

If all of the factors contained the largest possible values, the product would exceed 18 digits -- it would be 24 digits long. The programmer does not expect a product longer than 18 digits, because when some factors are very large, the others are supposed to be very small.

A size error in this case would be a check on the validity of the input data. In the event of a size error, the programmer will have the computer perform an analysis routine to find out which factors are not reasonable. So he has written:

```
COMPUTE X = A * B * C * D; ON SIZE
    ERROR, PERFORM FACTOR-ANALYSIS.
```

The alternative would be to analyze the factors before multiplying them -- but that would increase the time for calculating X every time. And an error will rarely, if ever, occur.

●●●

164 *The size error test is the only test that can be performed during an arithmetic calculation. It is built right into the arithmetic actions. Many other tests are possible, though, if they are done prior to an arithmetic statement or after it. For example, after an arithmetic result has been calculated, it can be tested to see if its value is zero. Or if its sign is plus or minus. Or if it is less than or greater than established limits. Tests such as these are simply written as separate IF sentences in the procedure.*

●●●

165 Arithmetic action 4. The raw result is moved from the work area to the finished result item. The move will be a numeric move if the receiving item is numeric, or an edit move if the receiving item is a report item.

●●●

166 *Let me emphasize that you do not write a MOVE statement to get the result into the finished result item. The compiler generates the instructions that are needed to get the result into that item.*

●●●

167 As you learned in the preceding course, there are nine basic formats for arithmetic statements. Four of those formats are illustrated below. In each of these statements, the finished result item is _____.

```
ADD A TO B.
SUBTRACT A FROM B.
MULTIPLY A BY B.
DIVIDE A INTO B.
```

●●●

B

168 In each statement shown in the preceding frame, items A and B are both used in the calculation, so both items must be elementary numeric items. For each statement, therefore, when the result is moved from a work area to the finished result item,
[a numeric move] [an edit move] will occur.

●●●

a numeric move

169 These statements illustrate the remaining five formats.

```
COMPUTE R = A + B * C.
ADD A, B, C, D, GIVING R.
SUBTRACT A, B, C, FROM D, GIVING R.
MULTIPLY A BY B, GIVING R.
DIVIDE A INTO B, GIVING R.
```

In each statement, the finished result item is R. In none of these cases is R used in the calculation; it may therefore be either an elementary numeric item or an elementary report item. The move from the work area to the finished result will be [a numeric move] [an edit move].

●●●

EITHER a numeric move OR an edit move (depending on the receiving item)

170 *It is important for you to have the effect of the GIVING clause well in mind. At this time, re-read the brief discussion of GIVING in the reference manual.*

Reading assignment: Arithmetic statements, GIVING option

●●●

171 Conversion may be required when the result is moved to the finished result item. This is true for both a numeric move and an edit move. Remember that the raw result will be either binary or packed-decimal. Conversion occurs when the usage of the finished result item is different from that of the raw result.

Suppose the raw result is a packed-decimal number. The finished result item is described by the entry below. Will conversion occur?

```
02  DEFLECTION
    PICTURE SV9(8), COMPUTATIONAL-3.
```

●●●

No (COMPUTATIONAL-3 usage means packed-decimal data format.)

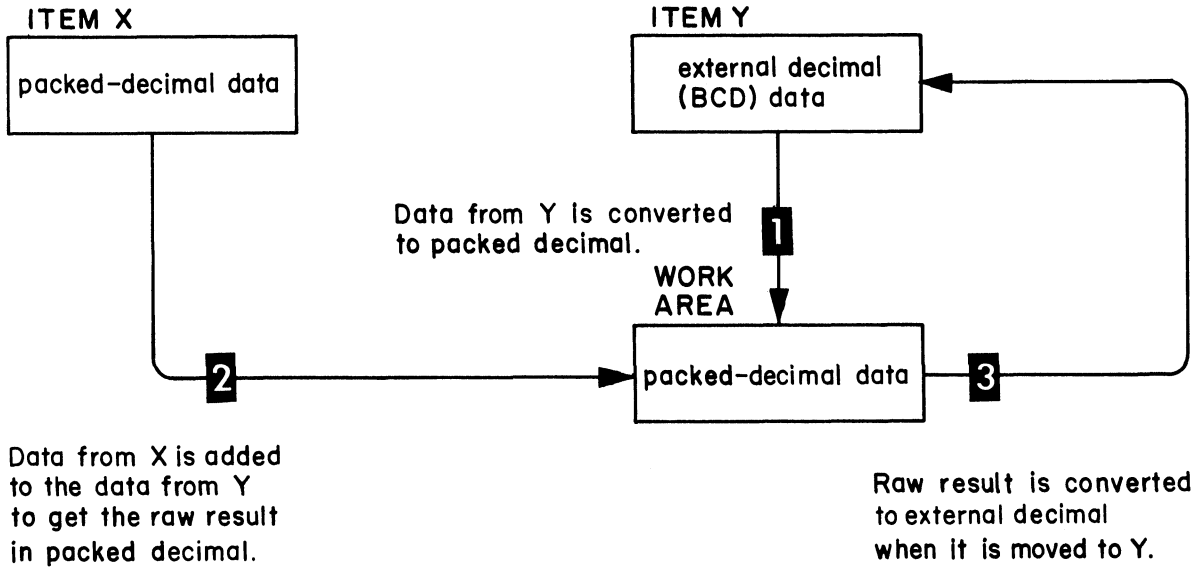
172 Conversion of the raw result will always occur when the finished result item is [an elementary numeric item] [an elementary report item].

•••

an elementary report item (All report items have DISPLAY usage.)

173 The conversion of the raw result may represent the second or third time conversion has been done in the process of executing an arithmetic statement.

This chart illustrates a process in which two conversions take place. The arithmetic statement being executed is "ADD X TO Y".



The programmer might also have written "ADD Y TO X". During the execution of this statement,

{ no conversion / one conversion / two conversions / three conversions } would have been required.

•••

one conversion (The data from Y would have to be converted to packed-decimal in order to add it to the data from X; but the raw result would not have to be converted because the finished result item, X, is a packed-decimal item.)

174 As you know, other actions may take place during the move to the finished result item. For one thing, a sign may be generated, corresponding to the sign of the result, or "no sign" bits may be placed in the result. Whether a sign or "no sign" bits are generated depends on _____.

●●●

whether there is an S in the picture of the finished result item

175 Decimal alignment is another action during the move, and as a result of decimal alignment, truncation or zero-fill (or both) may also occur.

Suppose that the raw result is 877v627594. What will the finished result be if the picture of the finished result item is (1) Z,ZZZ.99 and (2) S9(5)V99?

●●●

(1) bb877.62 (2) 00877v62 (In both cases there is truncation of low order digits; in the second case, there is also zero-fill of the high order positions.)

176 In the example given in the preceding frame, notice that the extra decimal places are just dropped, and not rounded off to the nearest penny. If the programmer had wanted to, he could have caused the raw result to be rounded before it was moved to the finished result item. "Rounded" means that the rightmost digit to be moved to the finished result item is increased by 1 if the digit to the right of it in the raw result is 5 or greater.

A raw result is 04v3819. The programmer has specified that the result is to be rounded. What will the finished result be if the picture of the finished result item is (1) 99 (2) 99.9 (3) 99.99 (4) 99.999 (5) 99.9999?

●●●

(1) 04 (2) 04.4 (3) 04.38 (4) 04.382 (5) 04.3819

177 In the preceding frame, the fifth picture contains just as many decimal places as the raw result. Thus, there is no digit to the right of the rightmost digit that will be moved. The finished result is the same as the raw result -- no rounding is possible. This situation illustrates that you should specify a rounded result only if there are extra digits at the right end of the raw result which will be _____ when the result is moved to the finished result item.

●●●

truncated (dropped)

178 *It is very easy to specify that the finished result is to be rounded.*

Reading assignment: Arithmetic statements, ROUNDED option

●●●

179 Shown below are two sets of entries. Which set of entries indicates the correct way to specify a rounded result -- or are both sets of entries correct?

(1)

```
    MULTIPLY QUANTITY BY PRICE,
        GIVING AMOUNT, ROUNDED.

77  AMOUNT, PICTURE S9(4)V99,
        COMPUTATIONAL-3.
```

(2)

```
    MULTIPLY QUANTITY BY PRICE,
        GIVING AMOUNT.

77  AMOUNT, PICTURE S9(4)V99,
        COMPUTATIONAL-3, ROUNDED.
```

●●●

Entry set (1) is correct. Entry set (2) is definitely not correct.

180 Carefully examine the formats of all arithmetic statements in the reference manual. The ROUNDED option can be used

{ in arithmetic statements that contain a GIVING clause.
in all arithmetic statements except COMPUTE statements.
in all arithmetic statements. }

●●●

in all arithmetic statements

181 In the reference manual, also observe where the word ROUNDED is written.

ROUNDED is always written

{ at the end of the arithmetic statement.
{ just ahead of the ON SIZE ERROR clause.
{ right after the name of the finished result item.

●●●

right after the name of the finished result item.

182 Here are two arithmetic statements which call for the same calculation. On a program sheet, rewrite these statements, adding ROUNDED in the correct place in each statement.

```
    COMPUTE PRICE = AVERAGE * .85.

    MULTIPLY AVERAGE BY .85, GIVING PRICE.
```

●●●

```
    COMPUTE PRICE, ROUNDED = AVERAGE * .85.

    MULTIPLY AVERAGE BY .85,
        GIVING PRICE, ROUNDED.
```

LESSON 5

183 *Our next topic is "condition tests". Here you will review some things you have learned about IF sentences in the earlier courses of this series, and then go on to learn about "compound" conditions and "nested" IF statements.*

Before you proceed, carefully review the formats of IF sentences and the five test conditions. These are summarized in the section on Procedure Division Entry Formats in the Writing Programs in COBOL reference handbook.

●●●

184 Check your ability to write both basic kinds of IF sentences. First, write a sentence which will cause the value of MAJORS to be increased by 1 provided that the value of AGE is more than 20. (Assume that the items called MAJORS and AGE have already been defined correctly in the Data division.) Use a program sheet for this.

●●●

```
IF AGE > 20, ADD 1 TO MAJORS.
```

185 On the same program sheet, write another IF sentence. This time, add 1 to MINORS if AGE is equal to or less than 20, in addition to adding 1 to MAJORS if AGE is greater than 20.

●●●

```
IF AGE > 20, ADD 1 TO MAJORS;
    ELSE, ADD 1 TO MINORS.
```

186 *Like most decisions, the sentence you were asked to write in the previous frame might have been written in various ways. In fact, the same processing could have been done in three IF sentences:*

```
IF AGE < 20, ADD 1 TO MINORS.
IF AGE = 20, ADD 1 TO MINORS.
IF AGE > 20, ADD 1 TO MAJORS.
```

Part of a programmer's task, though, is to reduce a procedure to its most simple and efficient sequence. It is less efficient, of course, to ask the computer to make three tests (as specified above) instead of one test (as in the solution printed in the preceding frame).

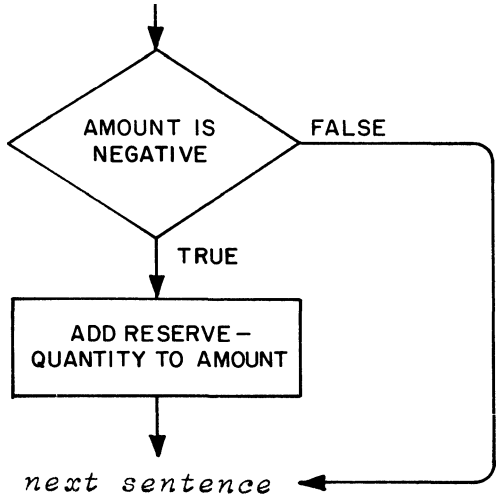
●●●

187 *It will be worth your time, also, to review the flow of control through IF sentences. If you are uncertain about it, re-read Flow of Control in the COBOL Program Fundamentals reference handbook.*

●●●

188 *When we chart the flow of control, we generally indicate the test condition and the procedural statements in the sequence in which they appear in the IF sentence. The "true" condition is represented as a continuation of the flow of control, while "false" is shown as a bypassing of certain statements. Below is an IF sentence and its equivalent flowchart, to illustrate the point.*

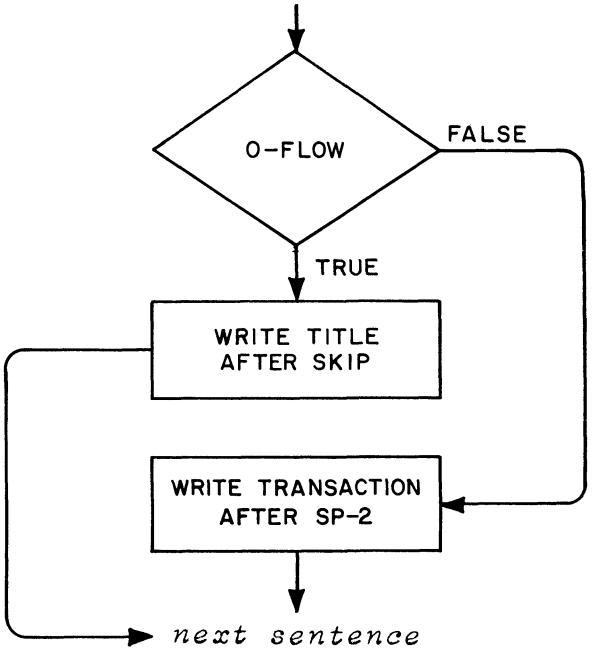
```
IF AMOUNT IS NEGATIVE,
    ADD RESERVE-QUANTITY TO AMOUNT.
```



●●●

189 *Whenever control comes to the word ELSE or OTHERWISE, it immediately goes on to the next sentence. Control flows to the statement after ELSE or OTHERWISE from the previous "false" condition.*

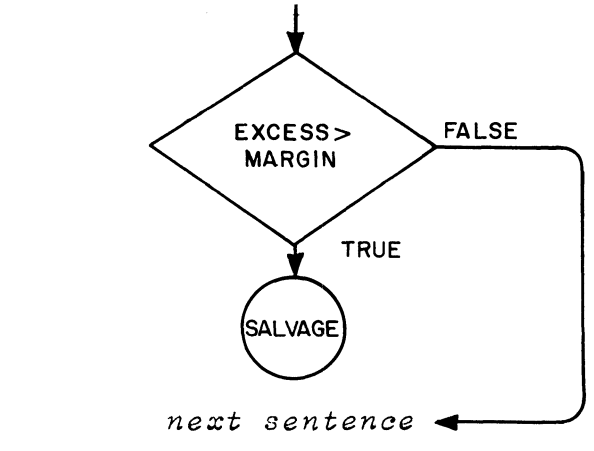
```
    IF O-FLOW, WRITE TITLE AFTER SKIP;
       ELSE, WRITE TRANSACTION AFTER SP-2.
```



●●●

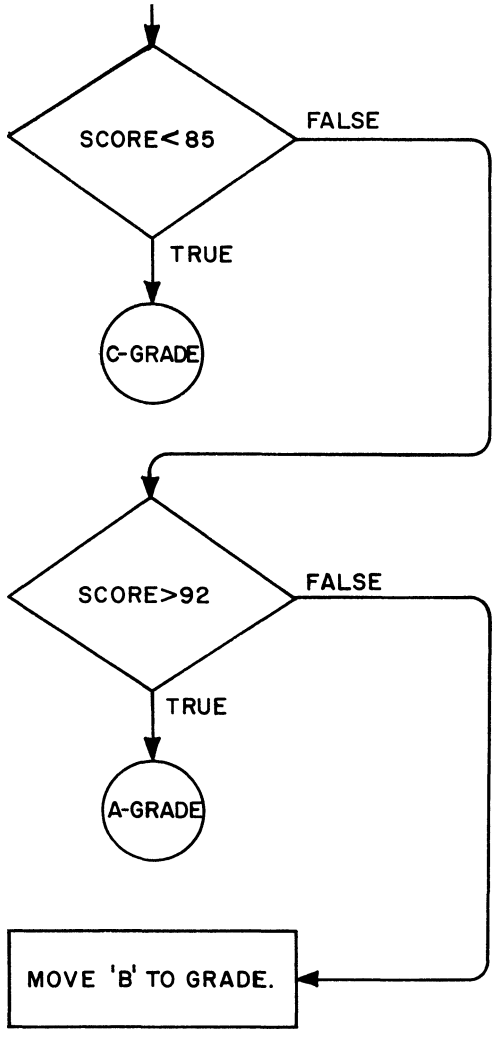
190 *A statement in an IF sentence may call for a branch. To avoid cluttering up the flowchart with too many lines, this is how a branch will be indicated in this book.*

```
    IF EXCESS > MARGIN, GO TO SALVAGE.
```



●●●

191 *This is the flowchart of a procedure which assigns a grade of "B" to a student whose score falls into the 85-92 range. On a program sheet, write the procedural statements that correspond to the flowchart. (Assume that all names used in the chart have already been defined.)*



●●●

```
IF SCORE < 85, GO TO C-GRADE.
IF SCORE > 92, GO TO A-GRADE.
MOVE 'B' TO GRADE
```

192 *The two tests made in the flowchart above can be combined into a single test condition, called a "compound" condition. Actually, the computer would still make two tests, but we would be able to combine them into one IF sentence.*

Reading assignment: Compound conditions

●●●

193 A compound condition is produced simply by tying together two or more test conditions with the words AND or OR. (The word NOT can be used as usual to give the opposite meaning to a condition.)

Compound conditions are written after the word IF. For example, IF ORDER < 500 AND REMAINDER > 100, GO TO FILL-ORDER. (Notice that the word IF is not repeated after the word AND.)

In this example, suppose that the value of ORDER happens to be 450, and the value of REMAINDER is 90; with these values, control

{will / will not} go to FILL-ORDER.

●●●

will not (The word AND means that both of the test conditions must be true; the value of REMAINDER must be more than 100).

194 The word OR means that the compound condition is true if either test condition is true. When the sentence below is acted on, suppose that both test conditions are true -- R-CODE is equal to 3, and R-NUMBER and P-NUMBER are unequal; in this event, control

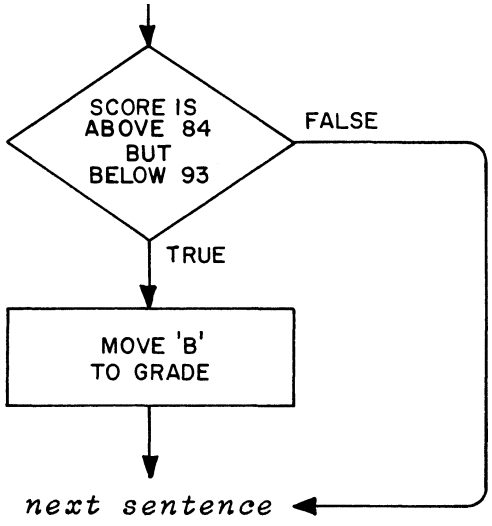
{will / will not} go to TOTALS.

```
IF R-CODE = 3, OR R-NUMBER IS NOT
    EQUAL TO P-NUMBER, GO TO TOTALS.
```

●●●

will (A compound condition containing OR is true if one test condition, or the other, or both are true.)

195 This flowchart shows how our earlier problem of assigning a "B" grade could be solved using a compound condition. Write the IF sentence that corresponds to the chart.



●●●

```
IF SCORE > 84 AND SCORE < 93,
    MOVE 'B' TO GRADE.
```

196 Here is another aspect of the grading problem. Write an IF sentence using a compound condition to cause the testing and branching described in the next paragraph.

"The value of SCORE is not valid when it is lower than 50. And a SCORE that is higher than 100 is also invalid. When a student's score is invalid, the program must branch to the procedure named INVALID-SCORE".

●●●

```
IF SCORE < 50 OR SCORE > 100,
    GO TO INVALID-SCORE.
```

197 One programmer wrote the following IF sentence, as his answer to the problem in the preceding frame. He found that it didn't work. Can you explain why?

```
IF SCORE < 50 AND SCORE > 100,
    GO TO INVALID-SCORE.
```

●●●

This sentence puts the computer into the logically absurd position of testing to see whether a number is simultaneously less than 50 and greater than 100. Both conditions cannot possibly occur at the same time, so the compound condition will always be false.

198 *Although compound conditions will often be very useful to you in programming, you should keep in mind that they cause the computer to make two or more tests -- and you want to avoid making two tests where one will do.*

Examine these two entries, one of which uses a compound condition, while the other does not. Determine what the difference is in the processing which the entries cause.

```
IF CONTRIBUTION = 100.00
    OR CONTRIBUTION > 100.00,
    DISPLAY 'GOOD SHOW' UPON CONSOLE.
```

```
IF CONTRIBUTION IS NOT LESS THAN 100.00,
    DISPLAY 'GOOD SHOW' UPON CONSOLE.
```

●●●

The entries produce the same end result, since "not less than" is the logical equivalent of "equal to or greater than". The second entry is preferable because it accomplishes the result by making one test instead of two.

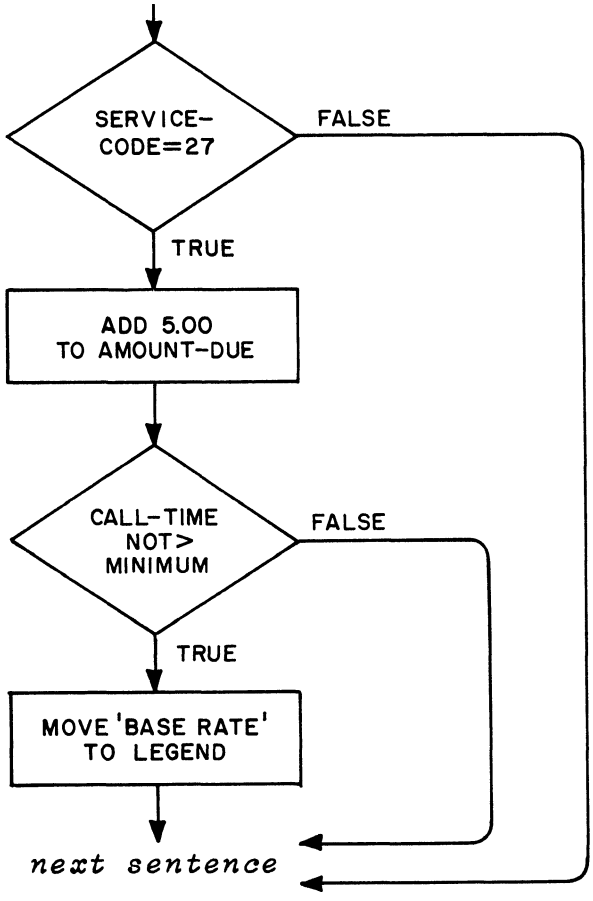
199 *Another kind of conditional sentence that is permitted in COBOL is one containing "nested" IF statements. The general idea of "nested" IFs is that one or more IFs appear within a sentence that begins with IF. Thereby it is possible to make a series of decisions based on the outcome of previous decisions, and to take different courses of action, all within one sentence.*

Reading assignment: Nested IF statements

●●●

200 *This sentence contains nested IFs. Study it and its flowchart. The flow lines on the chart should help you to see that the second decision is nested within the range of the first decision.*

```
IF SERVICE-CODE = 27,
    ADD 5.00 TO AMOUNT-DUE;
IF CALL-TIME NOT > MINIMUM,
    MOVE 'BASE RATE' TO LEGEND.
```



●●●

201 *The next 5 frames are based on the entry and flowchart printed in the previous frame.*

It is important to realize that the presence of a nested IF does not change the basic flow of control. Look at the first decision, and consider it to be just another IF sentence that does not contain ELSE or OTHERWISE. As will all such IF sentences, when the condition is "false", control goes to _____.

●●●

the next sentence

202 The next sentence { does / does not } begin right after the semicolon.

●●●

does not (The next sentence begins after the period, and therefore is not shown.)

203 If the first condition is "true", control proceeds to the next statement in sequence (ADD 5.00 to AMOUNT-DUE). After that, the next statement in sequence is _____.

●●●

IF CALL-TIME NOT > MINIMUM, MOVE 'BASE RATE' TO LEGEND

204 Again, this statement is treated just like a simple IF statement that does not contain ELSE or OTHERWISE -- even though it is part of an IF sentence. If the condition is "true", the MOVE statement written after the test condition is carried out. If the condition is "false", control goes directly to _____.

●●●

the next sentence

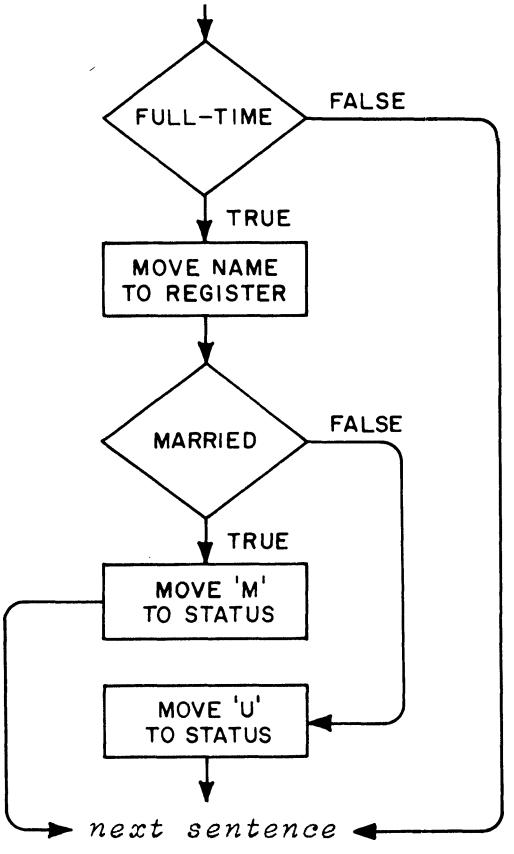
205 Especially important: The second condition is tested only if the first condition is found to be _____.

●●●

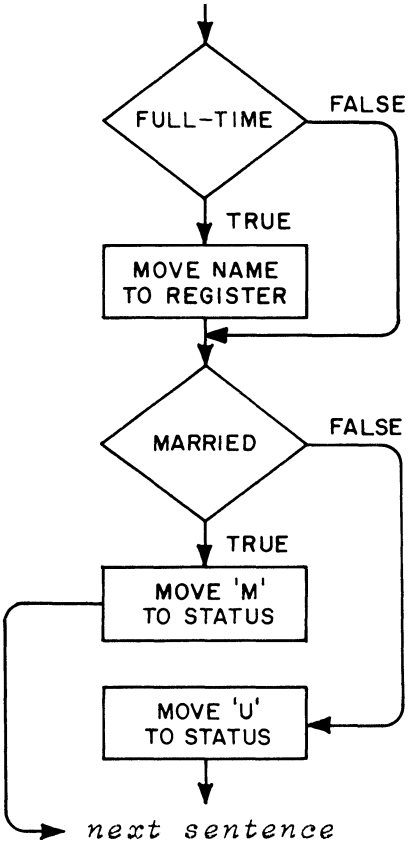
true

206 Don't make the mistake of thinking that nested IF statements are the same as a series of IF sentences. Below, side by side, are entries that are identical except for a single period. But observe the difference in the flow of control.

```
IF FULL-TIME,
MOVE NAME TO
REGISTER,
IF MARRIED,
MOVE 'M' TO
STATUS,
ELSE MOVE 'U'
TO STATUS.
```




```
IF FULL-TIME,
MOVE NAME TO
REGISTER.
IF MARRIED,
MOVE 'M' TO
STATUS,
ELSE MOVE 'U'
TO STATUS.
```



In the statements on the left, the second condition-name test is made only if the first condition is "true". On the right, the second test is made (when?) _____.

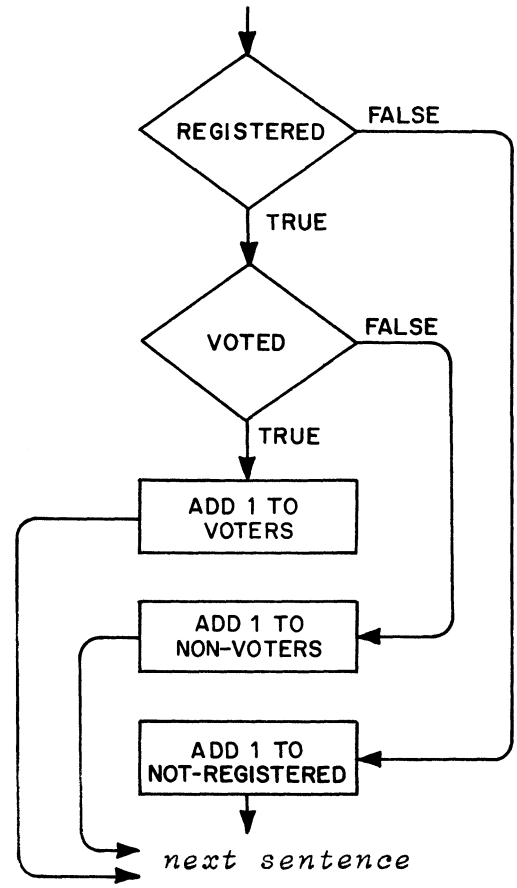
●●●

regardless of whether the first condition is "true" or "false"

207 The preceding frame illustrates that the rule about IFs that contain ELSE applies to nested IFs. The rule is: When control comes to ELSE, it goes to the next sentence. Thus, if you want control to leave a nested sequence and to go the next sentence, simply write ELSE, followed by whatever statements are to be carried out on a "false" condition.

Naturally, with nested IFs there will be more than one "false" condition. The rule here is: Control flows to the statement after ELSE from the last previous "false" condition that has not already been paired with an ELSE. This means that the ELSE statements for the first IF will come after the ELSE statements for the second IF.

The flowchart at the right illustrates this situation. It represents nested IFs, with each IF having an ELSE statement. The ELSE statement for the first condition is _____, while the ELSE statement for the second condition is _____.



●●●

ADD 1 to NOT-REGISTERED; ADD 1 TO NON-VOTERS

208 On a program sheet, write the single IF sentence that corresponds to the flowchart in the preceding frame. (Assume that all names have been defined. The decisions are condition-name tests.)

●●●

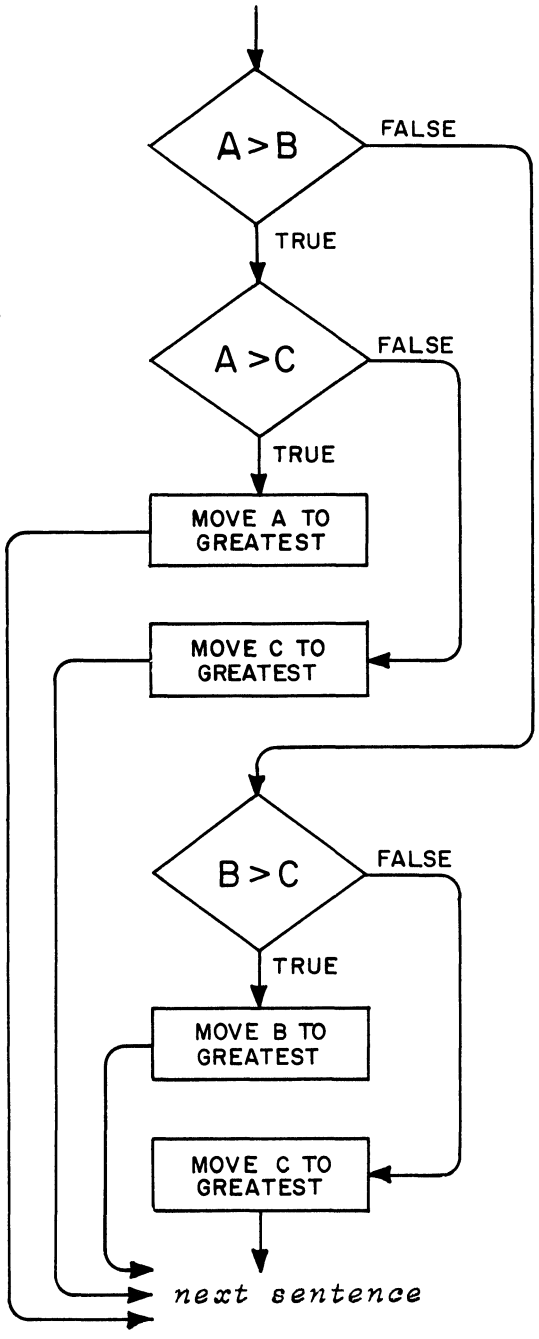
```
    IF REGISTERED, IF VOTED, ADD 1 TO VOTERS;
    ELSE, ADD 1 TO NON-VOTERS;
    ELSE, ADD 1 TO NOT-REGISTERED.
```

209 At the right is the flowchart of another IF sentence. This one is slightly more complex than the one in the previous frame -- but it is <u>just</u> <u>one</u> IF sentence nonetheless.

The flowchart represents the process of picking out the greatest of three numbers, A, B, and C. The three numbers have unequal values, for example, A might have a value of 25; B might be 10; and C might be 50. The process would determine that C (in this case) has the greatest value, and would move the contents of C to an item named GREATEST.

Analyze the flowchart to assure yourself that the process will work. You might try different values of A, B, and C, to make sure the highest value is the one that will be moved.

Then write the single IF statement that corresponds to the flowchart.



●●●

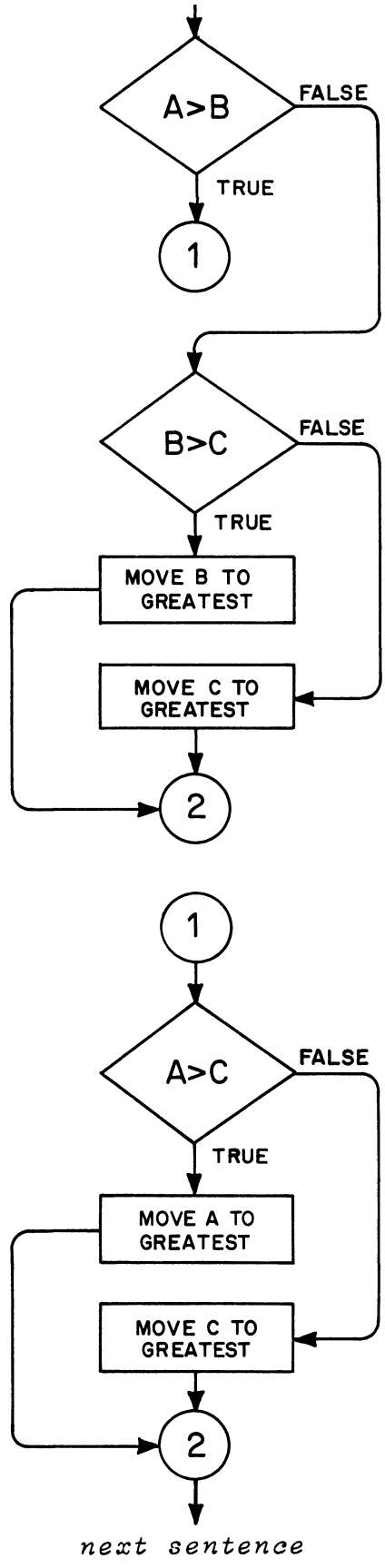
```
IF A > B, IF A > C, MOVE A TO GREATEST;
ELSE, MOVE C TO GREATEST;
ELSE, IF B > C, MOVE B TO GREATEST;
ELSE, MOVE C TO GREATEST.
```

210 *If you got lost somewhere along the way in our discussion of nested IFs, you may find it reassuring to know that any decisions made by nested IFs can also be made using common, ordinary, un-nested IFs. And sometimes it is better to do exactly that.*

Nested IFs can become devilishly complicated. Such statements are almost impossible for a reader to understand, and even harder for another programmer to change when program maintenance becomes necessary. What's more, even the original programmer is bound to have trouble debugging them if the results are not correct.

In all fairness to nested IFs, though, I think we should admit that complicated program logic remains complicated whether or not the IFs are nested. To illustrate this, I have taken the problem you worked on in the last frame (finding the greatest of three numbers), and written the solution without using nested IFs. The COBOL statements are printed below, and the flow of control through them is shown by the chart on the right.

```
    IF A > B, GO TO 1.
    IF B > C,
    MOVE B TO GREATEST;
    ELSE,
    MOVE C TO GREATEST.
    GO TO 2.
1.  IF A > C,
    MOVE A TO GREATEST;
    ELSE,
    MOVE C TO GREATEST.
2.  next sentence
```



●●●

211 *A good rule is: Keep your COBOL statements simple if you can. But if you can't make them simple, document them completely -- with explanations of the processing they do (possibly NOTEs in the Procedure division), and with flowcharts of the logic of the process.*

●●●

LESSON 6

212 *In Lesson 5, you wrote IF statements without paying any special attention to the method by which conditions are tested by the computer. Frankly, how the testing is done in most cases should not matter to you. Take the "class test" for instance; you might specify that a certain item is to be tested to see if its data is numeric. The test will be made and you will get "true" or "false" as your answer. That's all you need to know; you need not worry about how the computer was instructed to determine the answer.*

A major exception is the "relation test", which compares the values of two data items to determine whether the first value is greater than, equal to, or less than the second value. The comparison may be done in one of two rather different ways, depending on the types of data items that are compared. Two pairs of data items may contain precisely the same data, but if the item descriptions are different, the results of comparing them may be exactly opposite!

These facts make a strong case in favor of exploring how relation tests are carried out. One more fact makes the case even stronger: comparisons of some items are not even allowed! It is these facts about relation tests that we will discuss in this lesson.

●●●

213 *For a start, read what the reference manual has to say on the subject.*

Reading assignment: Relation test
(including the table of permissible comparisons)

●●●

214 *A simplification of the table of valid moves helped us in Lesson 1. Here is a similar table of valid comparisons. As in the earlier table, all numeric items have been combined, and figurative constants and sterling items have been omitted. A checkmark means the comparison is valid, while a shaded box means the comparison is not valid.*

VALID COMPARISONS

First Operand		Second Operand: Group	Second Operand: Elementary — Alphanumeric	Second Operand: Elementary — Report	Second Operand: Elementary — Alphabetic	Second Operand: Elementary — Numeric
Group		✓	✓	✓	✓	✓
Elementary	Alphanumeric	✓	✓	✓	✓	✓*
	Report	✓	✓	✓		
	Alphabetic	✓	✓		✓	
	Numeric	✓	✓*			✓

* Only whole numbers in external decimal code (BCD)

●●●

215 The entries in this table (unlike the table of moves) are symmetrical. That is, it really doesn't matter which item is the first operand and which is the second operand; if it is valid to compare A with B, it is also valid to compare B with A. Most combinations of items can validly be compared, such as an alphanumeric item with an alphabetic item. Which combinations of items cannot validly be compared?

●●●

It is not valid to compare (1) a report item with an alphabetic item, (2) a report item with a numeric item, and (3) an alphabetic item with a numeric item.

216 There is a good reason for not allowing these comparisons; report, alphabetic, and numeric items are by definition incompatible items.

See if you can develop the reasoning for yourself. It will help to recall that:

(a) Numeric items must contain _____ but must not contain _____.
(b) Alphabetic items must contain _____ but must not contain _____.
(c) Report items must contain _____ and may also contain _____.

●●●

(a) digits (but not) letters, special characters, or spaces
(b) letters or spaces (but not) digits or special characters
(c) special characters or spaces (and also) digits and certain letters

217 *The importance of all of this for us is twofold. First, don't try to write "forbidden" comparisons. The compiler analyzes the pictures of items in relation tests, and rejects any incompatible combinations. Second, recognize that computer comparisons are necessarily more restricted than comparisons that we personally can make. For example, we can see that $1.00 and 1v00 are equal values, but the computer can't see it that way.*

●●●

218 There are two types of comparisons:

(1) Alphanumeric (sometimes called "non-numeric")
(2) Numeric

Our chart of comparisons, this time showing the types of comparisons, is printed at the top of the next page. The chart shows that all comparisons are alphanumeric, except when an elementary _____ is compared with an elementary _____ item.

●●●

numeric; numeric

TYPES OF COMPARISONS

A = Alphanumeric
N = Numeric

First Operand		Second Operand: Group	Second Operand: Elementary: Alphanumeric	Second Operand: Elementary: Report	Second Operand: Elementary: Alphabetic	Second Operand: Elementary: Numeric
Group		A	A	A	A	A
Elementary	Alphanumeric	A	A	A	A	A*
	Report	A	A	A		
	Alphabetic	A	A		A	
	Numeric	A	A*			N

* Only whole numbers in external decimal code (BCD)

219 Alphanumeric comparison. Alphanumeric comparison is done in the same way that you might put words into alphabetical order. Values are compared character by character, proceeding from _____ to _____.

●●●

left; right

220 Comparing continues until two characters are found that are not the same, or until the ends of the items are reached. If two part numbers were being compared, and their values were 'AT9262' and 'AF8003', how many pairs of characters would the computer have to examine to know that the numbers are unequal?

●●●

Just two pairs, first A:A, then T:F

221 Characters are compared on the basis of the EBCDIC (Extended Binary-Coded Decimal Interchange Code) collating sequence. There is nothing unusual about this sequence, despite its formidable name.

Briefly, the collating sequence is as follows: (1) a blank space has the lowest value; (2) next lowest are the special characters; (3) then come the letters, A through Z -- A being lowest and Z being highest; (4) highest in value are the digits, 0 through 9.

Special characters are rarely involved in data which is compared, so we will not concern ourselves with the order of value of specific special characters.

According to the sequence of values, which has the greater value: '22A675' or 'R19451'?

●●●

22A675 (The digit 2 is greater than the letter R.)

222 'JONES' is { less than / greater than } 'SMITH'.

●●●

less than (J is less than S.)

223 'SMITHERS' is { equal to / greater than / less than } 'SMITHMAN'.

●●●

less than (Remember, the comparing proceeds character by character from left to right. In this case, the computer would find equal comparisons for the first five pairs of characters it examined. As the sixth pair, E and M would be compared; E is less than M, so 'SMITHERS' is the lesser value.)

224 Once in a while, you may find it necessary to compare items that have different lengths. The rule is that the shorter item is thought of as being filled out with blanks to the length of the longer item.

This rule is another way of saying that the comparison does not necessarily stop when the computer gets to the end of the shorter item. If all of the characters have been equal up to that point, the computer will look at the remaining characters -- if any -- in the longer item, and compare them with blanks. So, if the remaining positions of the longer item contain blanks, the items are equal; but if the remaining positions contain any characters, the longer item has the greater value.

An example will make the point clearer. Suppose we are comparing customer names of unequal lengths. The value of one is 'KARLOV' and the value of the other is 'KARLOVSKY'. Right up to the end of the shorter item, all characters are equal. But the comparison does not stop there; it proceeds to look at the next position of the longer item, and compares the character in that position with a blank. The items, then, are treated as if their lengths were equal and the shorter value were filled out with blanks, like this:

In this way, 'KARLOVSKY' is found to be greater than, not equal to, 'KARLOV'.

Let's try another example. What would be the result if 'MENDEZ' were compared with 'MENDEZbbbbbbbb'?

●●●

These values are equal, since the shorter item is treated as if it too were filled out with blanks.

225 Don't make the mistake of thinking that longer items are always greater except when they are filled out with blanks.

'SANDER' is { equal to / greater than / less than } 'SANDBLASTER'. Why?

●●●

greater than, because E > B when the fifth pair of characters is compared

226 Numeric comparison. Numeric comparison is done only when a _____ item is compared with a _____ item.

●●●

numeric; numeric

227 Differing usage of numeric items does not prevent a comparison of their values.

This means that a binary item { can / cannot } be compared with a packed-decimal item.

●●●

can

228 When usages are different, however, the computer will be instructed to convert one item or the other to make the usages the same. So, when a binary and a packed-decimal item are compared, the binary item is converted to packed decimal before the comparing begins.

This action is the same as the action of preparing data values for an arithmetic operation. Numeric comparison is the same as arithmetic in another respect: the computer can compare two packed-decimal numbers, or it can compare two binary numbers. Conversion is required whenever the usages of items are different, and whenever external-decimal items are compared.

As you know, when the computer must convert data codes, it takes additional instructions and more time to get the desired end result -- and we say that the object program is less efficient. It should be quite clear that the relative efficiency of the program depends mainly on the characteristics of the data arithmetic being processed -- and much less on the way a relation test or an arithmetic statement is written in the Procedure division.

Two numbers are to be compared, and you are in a position to influence the data codes of the items which will contain the numbers. Which of the three choices listed below would you recommend for the most efficient comparison? Which would be the least efficient? Why?

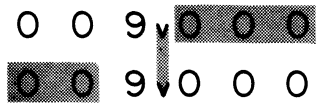
(a) Make both items packed decimal.
(b) Make both items external decimal.
(c) Make one item packed decimal, the other external decimal.

●●●

Choice (a) is most efficient because no conversion is required. Choice (b) is least efficient because both items must be converted for a numeric comparison.

229 Here is another action that will remind you of how arithmetic and numeric moves are done. The assumed decimal points of the values are aligned, and extra positions at either end of either value are filled out with zeros. The end result is that the sizes of the values are made the same. This action is done in work areas set up by the compiler.

If the values to be compared are 009 and 9v000, alignment and zero-fill will make the values look like this when comparison begins:



Of course, additional steps are used for such shifting and zero-filling.

In the statement below, suppose that the picture of AMOUNT is S999V99; what change would you suggest making in the statement to make its execution more efficient?

```
    IF AMOUNT > 500, GO TO CREDIT-CHECK.
```

●●●

Change the literal to 500.00. That makes the literal the same size as the data item, with the decimal points already aligned; therefore, no shifting or zero-filling is needed, and comparison can proceed at once.

The way in which the literal will be stored by the compiler will depend on the usage of AMOUNT. If AMOUNT is a binary item, that's the way the literal will be stored. If AMOUNT is packed decimal, the literal will be stored that way. The purpose, of course, is to avoid unnecessary conversions.

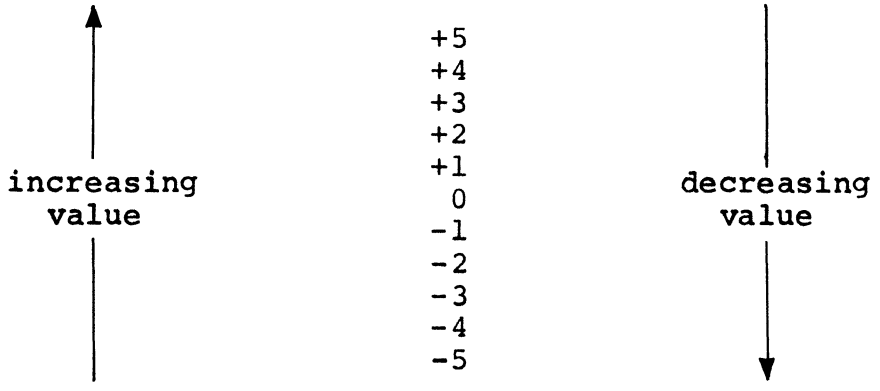
230 The actual comparison is based on two things: the sign of the number, and the magnitude of the number. Numbers that contain plus signs or no signs are considered "positive"; numbers that contain minus signs are "negative". The value zero is a special case -- zero has no magnitude and its sign, if any, is disregarded.

Therefore, the value -00v000 is { greater than / equal to / less than } the value +0000v0.

●●●

equal to

231 *You can think of numeric comparison as the process of locating a number's position along an "algebraic" number scale like the one shown below. The number that is higher on the scale has the greater value.*



Naturally, the scale can be extended endlessly in both directions, and fractional values fit into their appropriate places between the whole numbers shown here.

The scale shows that any positive number is greater than any negative number. Thus, -53v67 is { greater than / less than } +20v08.

●●●

less than

232 The greater the magnitude of a negative number, the lower its value on the scale. The value -4069v92 is { greater than / less than } -1v25.

●●●

less than

233 The converse is true of positive numbers: the bigger they are, the more they count. The value +4069v92 is { greater than / less than } +1v25.

●●●

greater than

234 *Programming considerations. When a number is stored in external-decimal code (BCD), we have an option of defining the item as alphanumeric or numeric. In this way, we are able to control the type of comparison that will be done with that number. In the next few frames, we will discuss some techniques of handling such numbers in comparisons; specifically:*

(a) the advantage of defining external-decimal numbers as alphanumeric.
(b) pitfalls to watch out for when defining numbers in this way.
(c) how to avoid unnecessary conversions when external-decimal numbers are defined as numeric.

●●●

235 Suppose that an item whose usage is DISPLAY contains a number, such as 36255. Such an item can be described as either an alphanumeric item or a numeric item. To make it an alphanumeric item, the item's picture must be _____. To make it a numeric item, the picture must be _____.

●●●

X(5); 9(5) or S9(5)

236

Notice the relation test in the above sentence. If the pictures of ACCOUNT-NUMBER and PREVIOUS-ACCOUNT-NUMBER are both 9(12), which type of comparison will occur?

●●●

numeric comparison

237 In the relation test shown in the previous frame, which type of comparison would have occurred if the picture of both items were X(12)?

●●●

alphanumeric comparison

238 External-decimal numbers are converted to packed decimal during [a numeric comparison] [an alphanumeric comparison].

●●●

a numeric comparison ONLY

239 Which is the more efficient way to define a department number which will be used in relation tests?

(1)
```
02 DEPARTMENT, PICTURE 999, DISPLAY.
```

(2)
```
02 DEPARTMENT, PICTURE XXX, DISPLAY.
```

●●●

Description (2) is more efficient because it eliminates the need for conversion to packed decimal.

240 *There are a few pitfalls that you must avoid when you define numbers as alphanumeric items. The worst pitfal exists when the numbers are signed (+ or -). Let's see why signed numbers can present problems in an alphanumeric comparison.*

It will help us to review how signs are stored in external-decimal items. A sign is represented by zone bits in the rightmost byte of the item. The value +7593 is in effect stored like this:

$$7593^{+}$$

The zone bits and the digit in the rightmost position are taken together, and treated as a letter or special character in an alphanumeric comparison.

The value 7593^{+} would be treated as 759C in an alphanumeric comparison.

●●●

241 In a numeric comparison, the values 125 and +125 are equal. In an alphanumeric comparison, the same values are unequal because _____.

●●●

the rightmost character of the second value would be taken to be a letter (specifically, +125 would be regarded as 12E).

242 In a numeric comparison, the sign is the first thing considered. Since an alphanumeric comparison proceeds character by character from left to right, the sign is the _____ thing considered.

●●●

last (Even so, it is not recognized as a sign, but as the zone bits of a character.)

243 If the values +236 and -435 were compared alphanumerically, +236 would be found to be { greater than / less than } -435.

●●●

less than (The first pair of characters compared would show that 2<4.)

244 In external decimal code, "$\overset{+}{2}$" is the same as "B" and "$\overset{-}{2}$" is the same as "K". Accordingly, in an alphanumeric comparison, -912 is { greater than / less than } +912.

●●●

greater than (The values are compared as 91K and 91B; K>B.)

245 *The moral of the story is that it is not wise to use an alphanumeric comparison when the values are signed numbers. Use a numeric comparison instead. However, defining numbers as alphanumeric items will still work well in the large number of cases in which unsigned external-decimal numbers are involved in comparisons; for example, part numbers, customer numbers, employee numbers, social security numbers, invoice numbers, etc.*

●●●

246 Another problem that arises in giving alphanumeric pictures to numbers is that alphanumeric data cannot be edited. In order to move the data to a report item, it must be numeric.

This problem is easily solved, by writing a second item description entry, this one containing a _____ clause and a numeric PICTURE clause.

●●●

REDEFINES

247 All things considered, you may decide in a particular case to define an external-decimal number as a numeric item only. Now you face the fact that the compiler will cause conversion to be done each time that item is used in relation tests, in arithmetic operations, and possibly in moves.

Such repeated conversion would make the program quite inefficient. In previous lessons, we discussed a technique by which repeated conversions can be prevented. Can you describe that technique?

●●●

Move the data from the external-decimal item to an internal-decimal item (or to a binary item -- whichever is appropriate) in working storage. Thereafter, use the name of the working-storage item in relation tests, arithmetic operations, and moves.

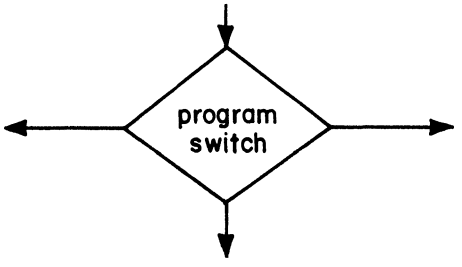
LESSON 7

248 *Still on the general subject of "decision making", we will now see how COBOL statements can be used to produce "program switches". Three different switching techniques will be explained. Along the way, you will be introduced to a new COBOL statement (ALTER) and a variation on an old COBOL statement (GO TO -- DEPENDING ON).*

●●●

249 A "program switch" is a point in a process, at which there is a choice of alternate paths. The choice of the path to take is determined by an earlier action.

On flowcharts, we generally show program switches as decision steps.

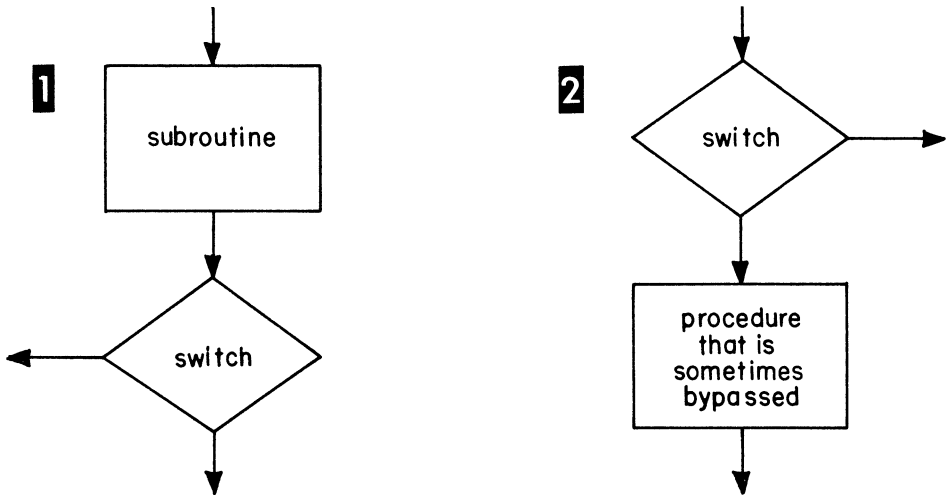


●●●

250 Unlike ordinary decisions (condition tests), in a program switch there is a delay between the action that determines which path will be taken and the actual flow of control down the chosen path. It's similar to a switch on a railroad track -- the switch is set before the train gets to it.

●●●

251 The main uses for program switches are (1) to create a "linkage" for subroutines, and (2) to "bypass" procedures at certain times while executing them at other times. The diagrams below illustrate these two uses.



Notice the difference. The switch is located after the subroutine, to cause control to flow back to somewhere in the main routine. The switch is located before the procedure that is sometimes bypassed, to cause control to flow _____.

●●●

around the procedure when bypassing is desired

252 At times, one switch may serve both purposes. Here is an example of such a switch. Our problem is to execute some procedures in this sequence: (1)A, (2)B, (3)C, (4)D, (5)B, (6)E. To form this sequence of procedures, we could write procedure B in two places -- but suppose that B occupies quite a few bytes of storage, so that writing it twice is out of the question. We will write procedure B only once and create a linkage to it using a program switch. The flowchart on the next page illustrates this solution.

To trace the flow of control, we start at A, then proceed to B. The switch must now permit control to flow down line 1 to C and D. No switch is shown after D -- but control must be made to flow along line 2 back to B; how can this be accomplished?

●●●

By writing an unconditional branch statement at the end of procedure D -- GO TO B.

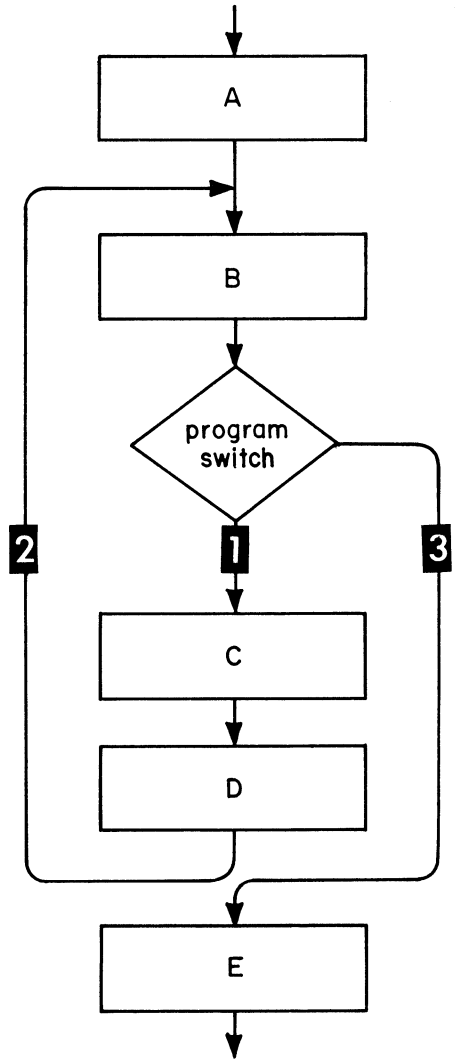
253 After procedure B has been executed for the second time, the switch must cause control to flow along line 3 to procedure E.

Let's review what this switch does. First, it provides the linkage which lets us use procedure B as a subroutine. It links to C after the first time through B, and links to E after the second time.

Second, it causes control to _____ procedures C and D after it leaves procedure B the second time.

●●●

bypass



254 We have made the point that the switch must be set to the desired path before control gets to the switch.

The switch must be set to line 3 sometime before control goes through B for the second time, but sometime after control goes _____.

●●●

down line 1 (In practice, we would set the switch to line 3 by a statement in procedure D, just before we say GO TO B.)

255 We may assume that this entire process is to be repeated more than once during the execution of the program. That being the case, we cannot leave the switch set to line 3. We must get it back to its line 1 setting; this might be done in procedure _____.

●●●

A is probably the best choice -- this sets the switch to line 1 just before control goes through B for the first time. (E is another possibility.)

256 Switching technique 1. The "IF" switch. This type of switch employs COBOL statements that you already know well -- an IF statement containing a relation test, plus a couple of MOVE statements.

The operation of the switch depends on the contents of a one-position item which has been defined in working storage by an entry such as:

```
77  INDICATOR, PICTURE X.
```

The switch tests whether this item has a certain value (for instance, 'R'); if so, a branch occurs to a specified procedure (in our case, the procedure name is E). Write an IF statement that will do this.

●●●

```
    IF INDICATOR = 'R', GO TO E.
```

257 To set the switch so that it will cause the branch to E, we must get an 'R' into INDICATOR. Write a statement to do this.

●●●

```
    MOVE 'R' TO INDICATOR.
```

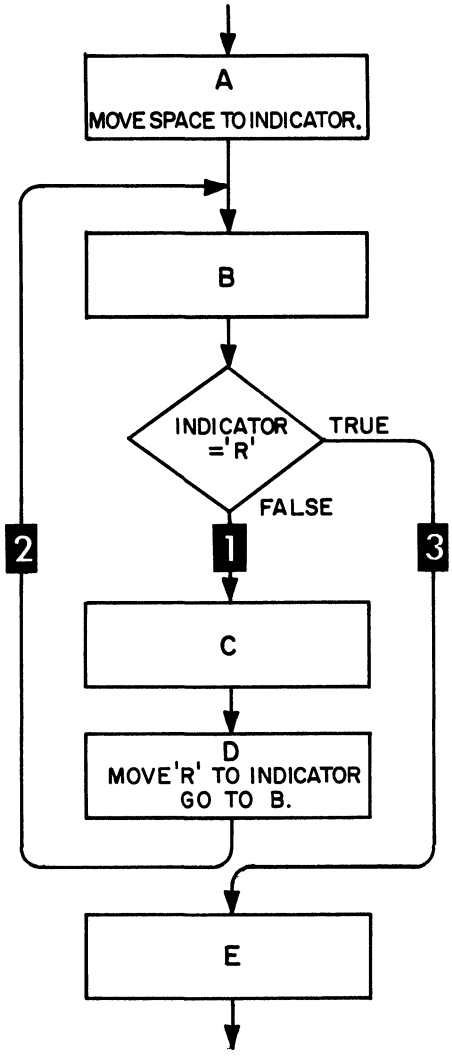
258 To allow control to flow to the next procedure in sequence (procedure C in our problem), the value of INDICATOR must be changed from 'R' to any other character or to a space. Write a statement to change the value to a space.

●●●

```
    MOVE SPACE TO INDICATOR.
```

259 *These three procedural sentences constitute the complete "switching mechanism". The only remaining question is, where should each of these sentences appear in the overall process? Turn back to the flowchart on the preceding page, and decide where each sentence should appear. Then check your solution with the one printed on the next page.*

●●●

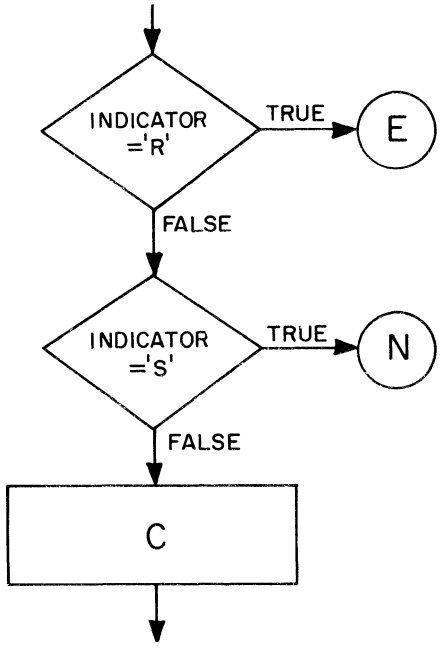

A
MOVE SPACE TO INDICATOR.
B
INDICATOR ='R'
TRUE
FALSE
2
1
3
C
D
MOVE 'R' TO INDICATOR
GO TO B.
E

260 Let's carry this type of switch one step further. Suppose that we want to go through our subroutine (procedure B) an additional time. You might imagine that in procedure M the need could arise to repeat the processing done in procedure B. This time, after B is executed, control should flow to procedure N.

To do this, we will write a second IF sentence immediately after the one that presently constitutes the switch. The new IF sentence will test the contents of INDICATOR for a different value -- say, 'S', and cause a branch to N if that value exists. The switch is now made up of a series of two IF sentences:

```
    IF INDICATOR = 'R', GO TO E.
    IF INDICATOR = 'S', GO TO N.
```

We can diagram the switch in this way:



Now the switch is prepared for a new setting. In procedure M, we must do two things: (1) change the value of INDICATOR to 'S'; (2) cause a branch to procedure B. Write the statements that do these two things.

●●●

```
    MOVE 'S' TO INDICATOR.
    GO TO B.
```

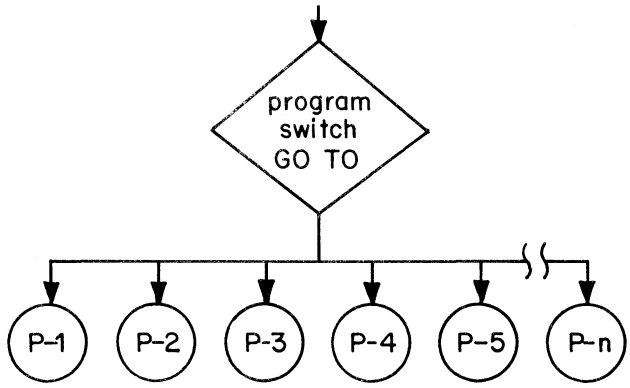
261 *To cause the "IF" switch to branch to still another procedure, we would just add another IF sentence to the series, and of course, another MOVE statement to put another value into INDICATOR.*

Envision, however, how this type of switch would appear if the subroutine were used ten or twelve times in the course of a program. The series of IFs would occupy half the lines on a program sheet, and there would be as many MOVEs. Worse still, this method forces the computer to make several relation tests, all but one of which will be "false", each time it leaves the subroutine. Our original problem required only one relation test, which didn't seem inefficient; but the thought of a dozen relation tests lined up in a row is enough to make you wonder, "Isn't there an easier way to do it?" Well, there is an easier way.

●●●

262 Switching technique 2. The "ALTER/GO TO" switch. In this type, the switch is a GO TO sentence which is altered by specifying the name of the procedure that you want to proceed to. In the problem we have been working, you could alter the GO TO sentence to proceed to C, or to E, or to N -- or to any number of other procedures.

No matter how many procedures you will go to, the switch still consists of just a single GO TO sentence. An "ALTER/GO TO" switch might be diagrammed like this:



In this diagram, I have tried to convey the idea that "any number can play" -- that is, you can make this GO TO branch to any one of countless procedures.

●●●

263 *To change the setting of the switch, you must write an ALTER statement.*

Reading assignment: ALTER statement
GO TO statement, Option 1

●●●

264 The first rule for this type of switch concerns the way in which the GO TO is written. The GO TO must be the only statement in a _____.

●●●

paragraph

265 The paragraph { must / may / must not } have a paragraph header.

●●●

must

266 The correct GO TO paragraph is [example 1] [example 2].

(1) SWITCH-1. GO TO C.

(2) SWITCH-1. GO TO.

●●●

BOTH are correct. *In the special case of the "ALTER/GO TO" switch, you are permitted to write a GO TO with no procedure name. The only requirement is that you must set the switch before control gets to it during the execution of the process; that is, you must alter the switch to proceed to a specific procedure before control gets to the switch.*

267 The second rule, then, is to write an ALTER statement. Write an ALTER statement which will cause the GO TO printed in the previous frame to branch to procedure E.

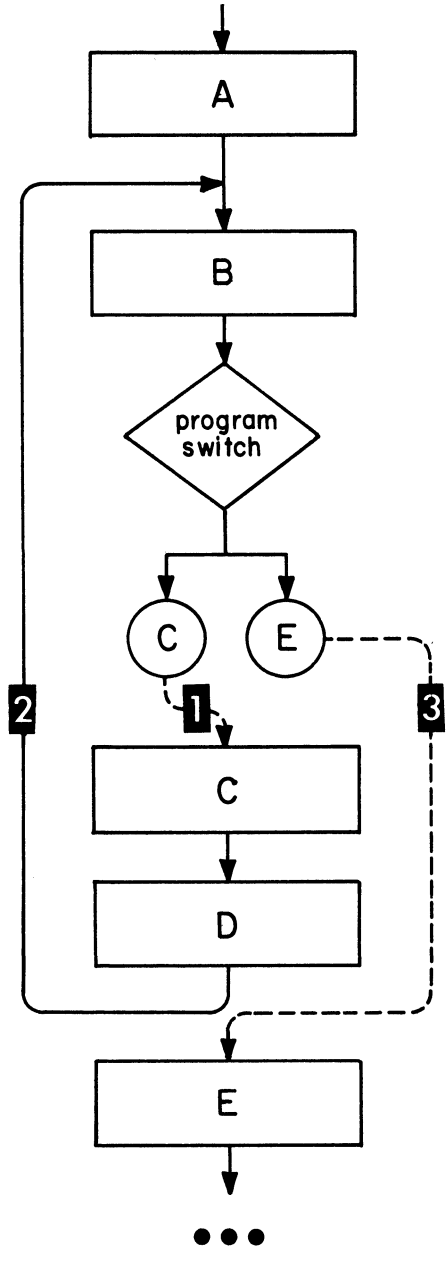
●●●

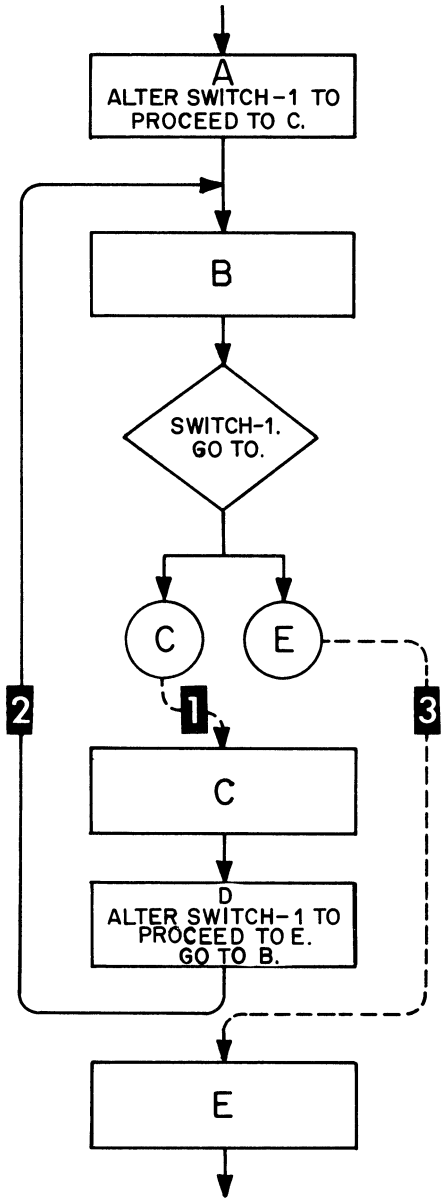
ALTER SWITCH-1 TO PROCEED TO E.

In "ALTER/GO TO" switches, no data values are tested, so no working storage items are defined. This really is "an easier way to do it!"

268 *Printed below is the same problem we solved using an "IF" switch, modified slightly to reflect the fact that an "ALTER/GO TO" switch must always specify an unconditional branch to some procedure. (In other words, there is no "fall through" to the next sequential procedure; to get to C in this problem, the GO TO must be altered to proceed to C.)*

Decide where the GO TO and ALTERs belong on the flowchart. Then turn the page to check your solution.





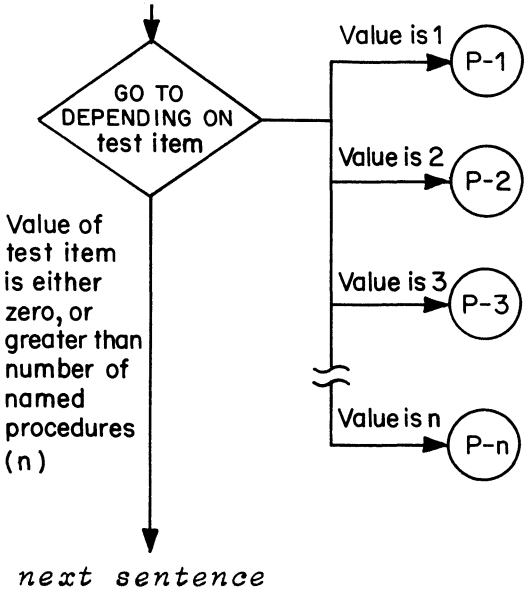
269 *"ALTER/GO TO" is an efficient type of program switch, largely because the computer is not required to make any decisions in carrying it out. The GO TO paragraph is converted to an unconditional branch instruction by the compiler. From each ALTER statement, the compiler generates an instruction to change the address in the branch instruction. "ALTER/GO TO" is the basic type of switching facility which is built into the COBOL language; it is undoubtedly the type of switch you will employ most often in your programs.*

●●●

270 Switching technique 3. The "GO TO/DEPENDING ON" switch. This type of switch is similar to the "IF" switch in that branching is based on the value of a data item. The switch lists the names of procedures to which control may branch. If the value of the data item is 1, control will go to the first procedure named in the list; if the value is 2, control goes to the second named procedure; if 3, it goes to the third named procedure; etc.

Any number of procedures can be named in the GO TO/DEPENDING ON statement. If the value of the data item is zero, or if it is greater than the number of named procedures, control automatically goes to the next sequential statement.

Here is how we will diagram this type of switch. This diagram also summarizes how control flows through the switch.



●●●

271 Reading assignment: GO TO statement, Option 2

●●●

272 *A GO TO/DEPENDING ON statement is the equivalent of a series of IF statements, so once again you can see the close similarity to the "IF" switch you studied first. But it is easier to write than a series of IFs, and the instructions generated by the compiler will be somewhat more efficient than a series of relation tests would be.*

The test item on which branching depends must be
[an elementary numeric item] [a group numeric item]
[an elementary alphanumeric item] [a group alphanumeric item].

It cannot contain more than (how many?) _____ digits.

It must be a positive (or not signed) whole number, and its usage may be _____, _____, or _____.

●●●

an elementary numeric item ONLY
four
COMPUTATIONAL (binary); COMPUTATIONAL-3 (packed decimal);
DISPLAY (external decimal -- BCD)

Four digits will permit a 9,999-way branch, which is enough for most of us. In fact, one or two digits are generally adequate. COMPUTATIONAL usage gives the most efficient action, followed closely by COMPUTATIONAL-3; DISPLAY usage calls for the most converting when the statement is executed.

273 Assume that a numeric item has been defined, and that it meets the requirements of the GO TO/DEPENDING ON statement. This series of IF statements is based on testing that item. Write the GO TO/DEPENDING ON statement which will cause the same branches as the series of IFs.

```
    IF JOB = 1, GO TO MANAGER.
    IF JOB = 2, GO TO ANALYST.
    IF JOB = 3, GO TO PROGRAMMER.
    IF JOB = 4, GO TO OPERATOR.
```

●●●

```
    GO TO MANAGER, ANALYST, PROGRAMMER,
       OPERATOR, DEPENDING ON JOB.
```

274 *As the previous frame implies, the use of GO TO/DEPENDING ON is not limited to program switching. It is a way of analyzing any code number and causing control to branch to procedures that correspond to various values of the number.*

In some cases, the code number is "made to order". For instance, if processing of input data is varied depending on the month in which a transaction occurred, and if the months are coded using the numbers 01 through 12, then we can easily specify that control is to GO TO twelve procedures DEPENDING ON MONTH. By the way, the procedure names do not all have to be different; if the processing for March and April is the same, we can simply make the third and fourth procedure names the same.

At other times, it will be necessary to derive the value of the test item by means of some calculation. Suppose a program processes transactions for the years 1961 through 1966, in a different way for each year; the year in which the transaction took place is coded as 61, 62, etc., in the input record. What calculation would make the value of YEAR suitable to use as a test item in a GO TO/DEPENDING ON statement?

●●●

SUBTRACT 60 FROM YEAR. (This makes the values of YEAR 01 through 06.)

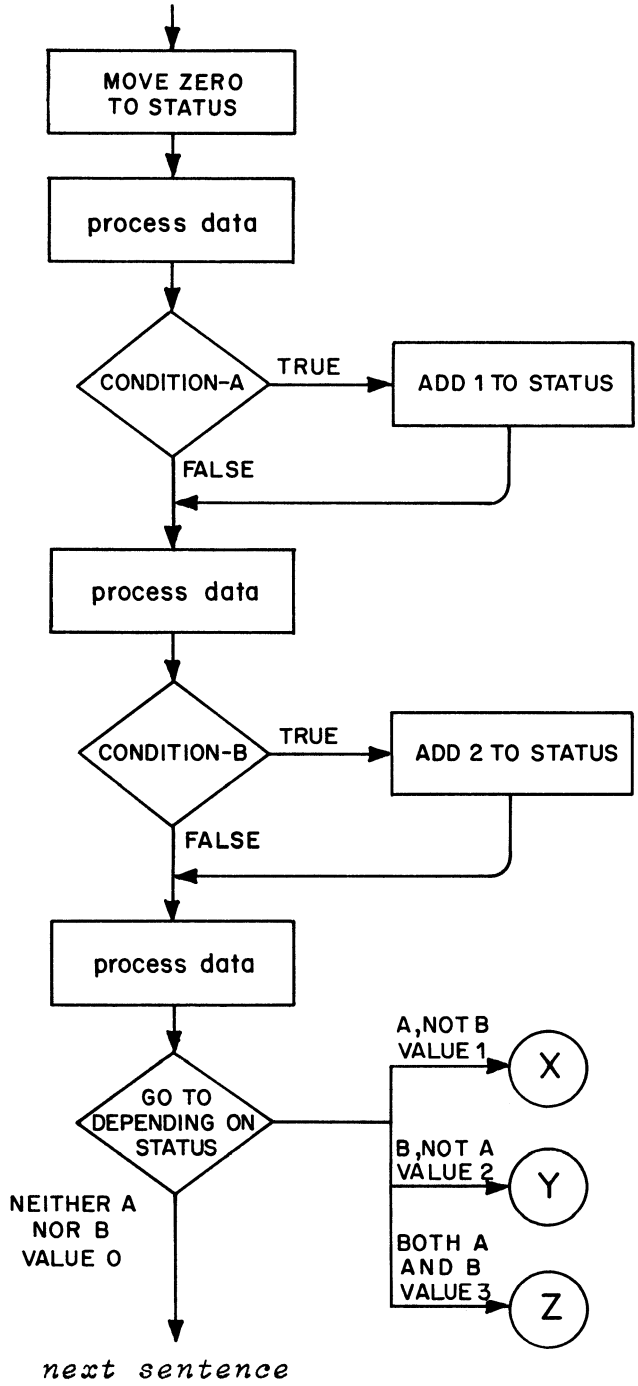
275 *When GO TO/DEPENDING ON is used as a program switch, it is necessary to set up an elementary numeric item in working storage to serve as the test item. To set the switch, change the value of the test item to a number corresponding to the desired branch.*

The test item value can be changed by moving in a new value, as we did with the "IF" switch. Another way of changing the value is by adding digits to the test item -- which would make the final value of the test item equal to the sum of a combination of values; this method is especially appropriate when the path to be taken at the switch depends on a combination of actions taken at separate points in a process, rather than on a single action.

Say that two events, A and B may occur sometime during a process. When we get to the switch, we want to go in four different directions depending on whether both A and B, only A, only B, or neither A nor B occurred. We can set the value of the test item to zero to begin with, add 1 to it if A occurs, and add 2 to it if B occurs. When control reaches the switch, the value of the test item will be zero if neither event occurred, 1 if only A occurred, 2 if only B occurred, and 3 if both events occurred.

●●●

276 *This diagram shows the logic of the switching method discussed in the preceding frame. Notice that this kind of application could not very well be done with an "ALTER/GO TO" switch -- there would be no point in altering the switch when testing condition A, because you can't know if condition B will occur later; and at condition B, how can you tell whether condition A occurred? Adding to the value of the test item in a "GO TO/DEPENDING ON" switch solves this problem quite handily.*



●●●

277 Write the description of a working storage item named STATUS which could be used as the test item in the process diagrammed on the opposite page.

•••

```
77  STATUS, PICTURE S9, COMPUTATIONAL.
```

278 Write the GO TO/DEPENDING ON statement which acts as a program switch in the diagram on the opposite page.

•••

```
    GO TO X, Y, Z, DEPENDING ON STATUS.
```

LESSON 8

279 *Two of the main uses of program switches are to create linkage to subroutines and to bypass procedures under certain conditions. These things can also be done by using the PERFORM verb.*

While it is simpler to write a PERFORM statement than to write a switch, it is a less efficient way to accomplish the result because more instructions are generated in the object program. When you use PERFORM, you are "letting George (the compiler) do it". There is a price to be paid in extra bytes of storage needed, because the compiler has to allow for every possible event -- such as the possibility that while you are performing one subroutine, you might decide to go off and perform a second subroutine, and then come back to the first one.

In addition, as you will learn in this lesson, PERFORM can be used for operations other than subroutine linkage. Most importantly, it can be used to control the number of times a loop is executed.

To sum up, PERFORM is a convenient and versatile part of the COBOL language. It allows you to set up subroutines and to control loops with a minimum of planning and writing. However, you can achieve these same functions with program switches; "switching" takes more planning and more writing, but results in fewer object program instructions.

●●●

280 *You have previously studied the format of the "simple" PERFORM. Before we look at the other options, let's review what you already know.*

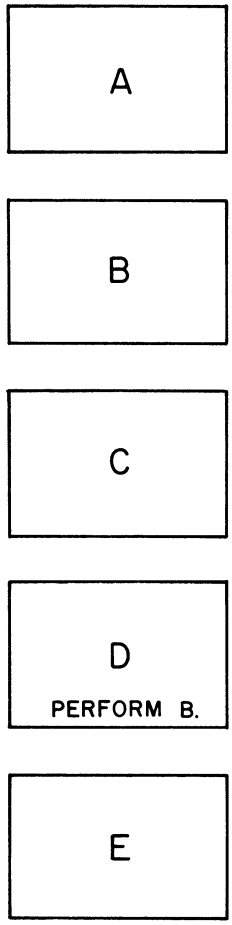
Reading assignment: PERFORM statement (Option 1)

●●●

281 Check your knowledge of how to apply the PERFORM statement. Take the problem we worked on in the preceding lesson: we have five procedures (A, B, C, D, and E) and we wish to execute procedure B twice. The desired order of execution is A - B - C - D - B - E.

On a piece of scratch paper, draw a series of five boxes to represent these procedures. Then decide what PERFORM statement is needed to cause procedure B to be executed a second time, and where that statement should be written.

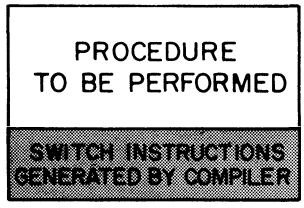
●●●



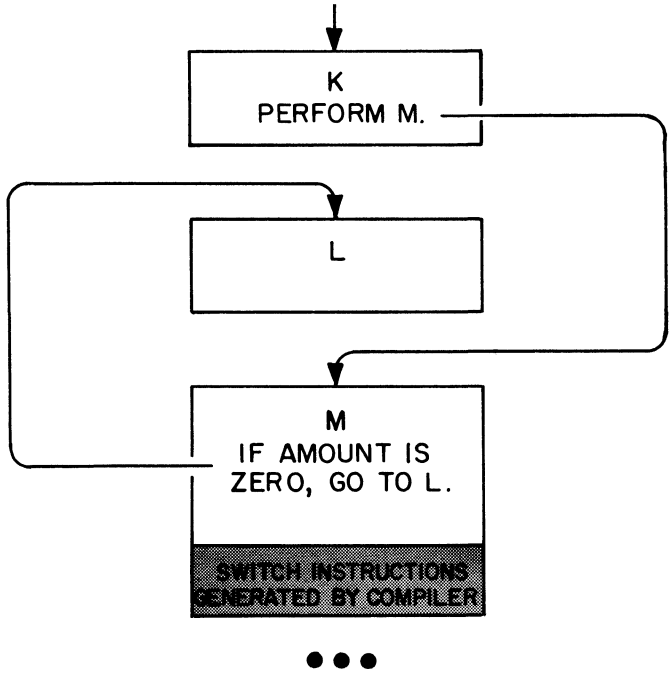
In this solution, I have written "PERFORM B" as the last statement in procedure D. Instead, I might have inserted a new procedure, containing only the PERFORM statement, between procedures D and E.

282 The statement "PERFORM B" sets up a switch at the end of procedure B, as well as instructions to set and reset the switch appropriately, and to cause control to branch to procedure B. The PERFORM statement does everything that we did by using program switches in the last lesson.

This diagram illustrates the location of the switch that is generated by the compiler.



When you "activate" a PERFORM, you must let control flow through the switch generated by the compiler. If you branch out of a procedure that is being performed, and fail to pass through the compiler's switch, you "mess up" the switch by not permitting it to be reset. This means that the IF sentence in the chart below is { allowed / not allowed }.

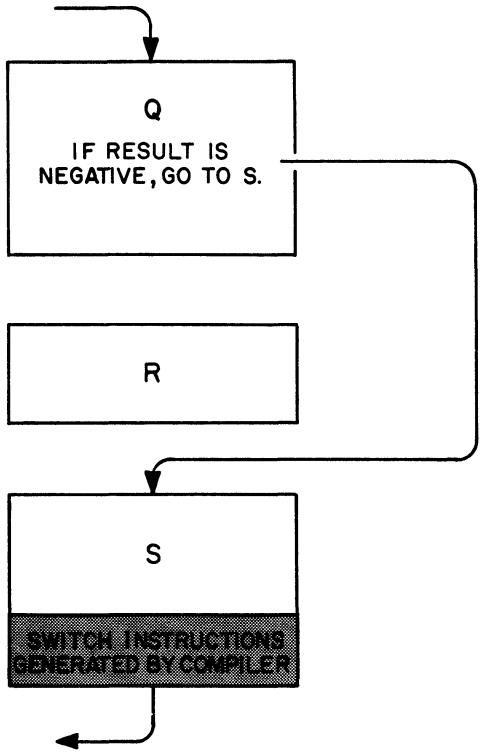


●●●

not allowed

283 Suppose that "PERFORM Q THRU S" is being executed. This chart shows the flow of control when RESULT is negative.

Is the branch away from procedure Q permissible?

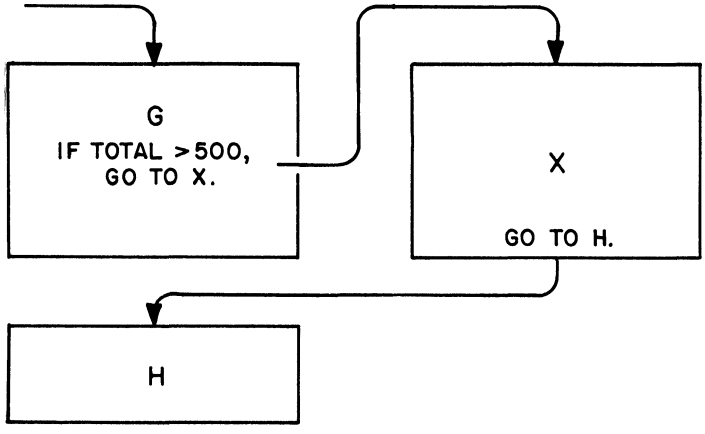


●●●

Yes. In this case, control bypasses some statements that are within the "range" of the PERFORM statement, but the flow of control is satisfactory because it does pass through the switch instructions generated by the compiler.

284 In this instance, the activated PERFORM statement is "PERFORM G THRU H".

When TOTAL exceeds 500, the programmer wants control to flow outside the range of the PERFORM, to procedure X, and then to return to H.



Can this be done? Also, where will the generated switch instructions appear?

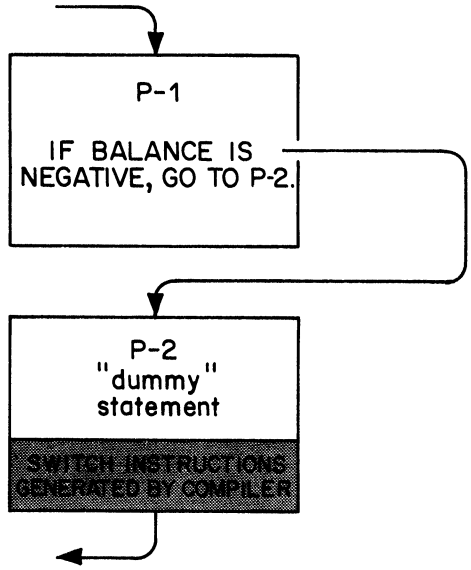
●●●

This can be done. Control will flow through the generated switch instructions, which will appear after procedure H.

285 *In the situations shown in the last two frames, control branched away from a procedure that was being performed, but then returned to execute more statements within the range of the PERFORM and to flow through the switch instructions.*

Imagine a slightly different situation, however. Under certain circumstances, you want to branch away from a procedure that is being performed, and to flow through the switch instructions, without executing any more statements in the range of the PERFORM. Suppose, for example, that you want to perform procedure P-1, but to leave P-1 and return to the statement after the PERFORM statement if BALANCE is negative. In order to do this, you must set up another paragraph (we'll call it P-2) which contains a "dummy" statement. The "dummy" statement will perform no operation, but it will be followed by the switch instructions generated by the compiler. When BALANCE is negative, branch to P-2; in this way, no more real statements will be executed, and control will flow through the switch as required. To get the switch instructions to be generated after P-2, you need to write "PERFORM P-1 THRU P-2" instead of just "PERFORM P-1".

This chart shows the flow of control when BALANCE is negative.



The "dummy" statement that is used in this kind of situation is EXIT.

Reading assignment: EXIT statement

●●●

286 An EXIT statement must appear in a separate _____.

●●●

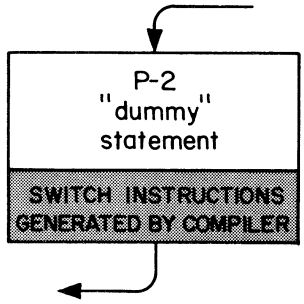
paragraph

287 Other statements { must / may / must not } appear in the same paragraph.

●●●

must not

288 On a program sheet, write the paragraph that corresponds to this block:



●●●

```
P-2. EXIT.
```

If you wrote more than this, you wrote too much. Remember that the compiler generates the "switch" instructions automatically when you write "PERFORM P-1 THRU P-2" elsewhere in the Procedure division.

289 *The three other formats of the PERFORM statement are used when it is desired not only to set up a linkage to a routine, but also to control the number of successive times the routine is executed. Usually, these formats apply when "looping" is desired, that is, when a process is to be repeated until a terminal condition prevails.*

One of these formats lets you specify the number of times a procedure is to be performed.

Reading assignment: PERFORM statement (Option 2)

●●●

290 There are few instances in which you need to specify a literal number of repetitions of a procedure, such as 2 or 10 or 25 times. But suppose you have a job in which you are printing shipping labels, and you want to produce exactly five labels for each customer. You have written a procedure named PRINT-LABEL, which prints one label. What PERFORM statement would you write to get five labels?

●●●

```
PERFORM PRINT-LABEL 5 TIMES.
```

291 More often, you will want to repeat a procedure a certain number of times for one record, and a different number of times for another record, depending on the value of some data item in the record.

Let's take such a case. Suppose that in processing late payments, a penalty is to be imposed, based on the number of days the payment is late, and compounded daily. For each payment, we may first figure out from the current date and the payment date, how many days it is late; let's assume that we have figured this out for a payment, and that the result is called DAYS-LATE.

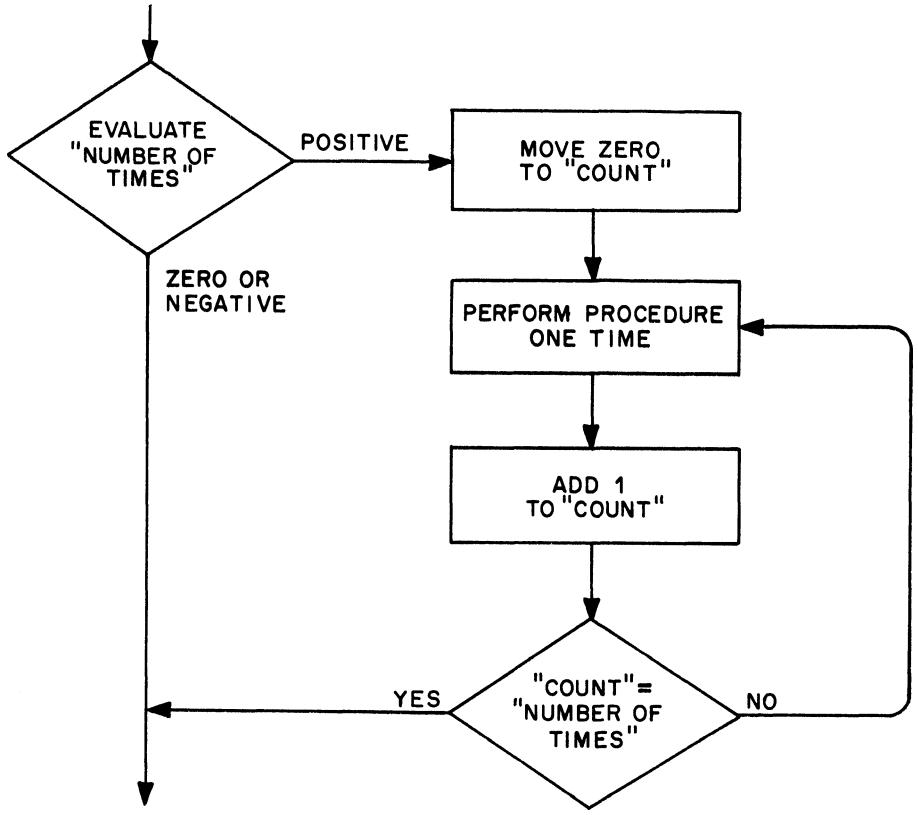
On a program sheet, write the entry that will cause a procedure called PENALTY-CALCULATION to be repeated a number of times equal to the value of DAYS-LATE.

●●●

```
PERFORM PENALTY-CALCULATION
        DAYS-LATE TIMES.
```

292 *This diagram shows how the "TIMES" option of the PERFORM statement works. It will be helpful to examine the diagram briefly, and even more helpful to return to it later to compare this option to the other options of PERFORM. In the diagram, the term "number of times" refers to the value of the data name or literal that precedes the word TIMES in the PERFORM statement, and "count" refers to a tally item set up by the compiler. (Incidentally, this diagram and the others you will see later in this lesson merely represent the chain of reasoning followed in executing PERFORM statements, and are not meant to show the exact series of instructions generated by the compiler.)*

LOGIC DIAGRAM OF "PERFORM/TIMES" STATEMENT



●●●

293 The logic diagram shows that the procedure is not performed at all if the "number of times" is _____ or _____.

●●●

zero, negative

This is a very useful feature. It amounts to a sort of "automatic program switch" which determines whether to perform or bypass the procedure.

294 Also notice in the diagram that the value of "count" is increased and compared with "number of times" { before / after } the procedure is performed.

●●●

after

295 Apply the logic diagram to the execution of the statement below. Suppose that the value of NUMBER-OF-MONTHS is 12.

Inasmuch as "count" is increased and tested <u>after</u> each performance of the procedure, INTEREST-COMPUTATION will actually be performed

{ 11 times / 12 times / 13 times }

```
PERFORM INTEREST-COMPUTATION
    NUMBER-OF-MONTHS TIMES.
```

●●●

12 times

Shortly, you will learn that in the remaining two options of PERFORM, the condition is tested <u>before</u> the procedure is performed, which of course, will affect the number of performances.

296 *Very often, we need to keep repeating a loop until we get a certain result or until a certain condition prevails - regardless of whether it takes one or one hundred performances. Options 3 and 4 of the PERFORM statement give us this type of loop control.*

Reading assignment: PERFORM statement (Option 3)

●●●

297 Consider this example of a PERFORM/UNTIL statement.

```
PERFORM ALLOCATION UNTIL RESOURCE IS
    ZERO OR RESOURCE IS NEGATIVE.
```

If the value of RESOURCE is 5000v00, will ALLOCATION be performed 5,000 times?

●●●

Not necessarily. It all depends on how the value of RESOURCE is changed by the ALLOCATION procedure. Let's say that the value of RESOURCE is reduced by 1000 each time ALLOCATION is performed; in that event, the procedure will be performed five times.

298 Look again at the PERFORM statement in the previous frame. What will happen if the original value of RESOURCE is 5000v00 <u>and</u> the value is <u>not</u> changed by the ALLOCATION procedure?

●●●

The ALLOCATION procedure will be repeated endlessly.

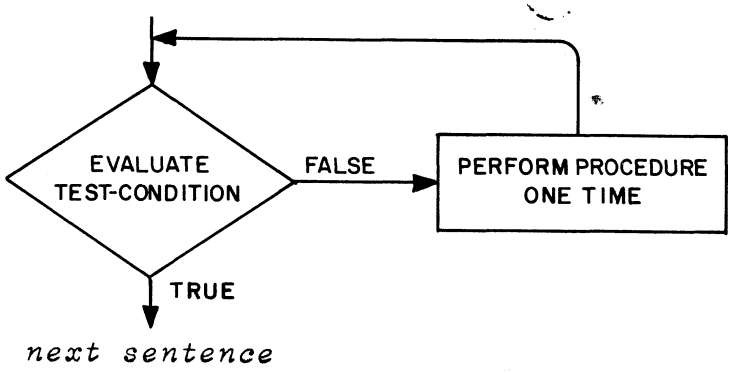
299 You can avoid the problem of the "endless loop" in a PERFORM/UNTIL statement by following this rule:

In the procedure to be performed, you { must / may / must not } change the value of the data whose condition is tested in a PERFORM/UNTIL statement.

●●●

must

300 LOGIC DIAGRAM OF "PERFORM/UNTIL" STATEMENT



●●●

301 Note that the test-condition is evaluated before the procedure is performed. If the test-condition is true to begin with, the procedure will

{ be performed only one time
not be performed at all
be repeated endlessly }.

●●●

not be performed at all

302 Apply the logic of "PERFORM/UNTIL" to this case. You want to perform a certain process for each part number between 31 and 74, inclusive. To initialize the procedure, you move 31 to a control number. The last statement in the procedure to be performed increases the control number by 1. You write the following PERFORM statement, but when you test your program you discover that only part numbers 31 through 73 are processed! Can you explain why you got this result, and what you can do to correct it?

```
    PERFORM PART-PROCESS UNTIL
        CONTROL-NUMBER = 74.
```

●●●

Control goes to the next statement as soon as the control number is found to equal 74, before PART-PROCESS can be performed for part number 74. You can correct this by changing the UNTIL clause to either "...UNTIL CONTROL-NUMBER = 75" or "...UNTIL CONTROL-NUMBER >74".

303 *So far, you have studied three options of the PERFORM statement: the simple PERFORM, PERFORM/TIMES, and PERFORM/UNTIL. The fourth option, the "PERFORM/VARYING" statement, is a little more complicated. Actually, it is quite similar to "PERFORM/UNTIL" -- except that the PERFORM does more than test for a terminal condition; it also moves an initial value to a data item and increases the value of the item by a specified amount each time the procedure is performed.*

In the most complicated form of the PERFORM/VARYING statement, the values of three data items are initialized and increased, and three conditions are tested. In effect, we can have a loop within a loop within a loop. For the purposes of this course, we will study only the least complicated form of the statement -- in which just one condition is tested and one item is initialized and increased. As you read the reference manual, therefore, skim lightly over the paragraphs and flowcharts that pertain to statements in which more than one data item is varied.

Reading assignment: PERFORM statement (Option 4)

●●●

304 One of the actions of the PERFORM/VARYING statement is to change the value of an item which we will call the "base item". This base item is usually involved in the processing done by the procedure that is being performed. In this statement, the base item is _____.

```
PERFORM RETIREMENT-COMPUTATION
    VARYING AGE FROM AGE-WHEN-HIRED
    BY 1 UNTIL AGE = 65.
```

●●●

AGE (The name of the base item always follows the word VARYING.)

305 Another action is to set the initial value of the base item. The initial value is specified in the FROM clause. What is the base item of this PERFORM statement, and what will be its initial value?

```
PERFORM DEPRECIATION, VARYING VALUATION
    FROM SALVAGE BY RATE,
    UNTIL VALUATION > COST.
```

●●●

The base item is VALUATION, and its initial value will be the value of SALVAGE.

306 After the procedure has been performed, the value of the base item is increased by a specified amount. Either the literal amount or the name of an item containing the amount is written after the word

{ UNTIL / BY / FROM / VARYING }.

●●●

BY

307 The last clause of a PERFORM/VARYING statement is an UNTIL clause. Just as in a PERFORM/UNTIL statement, the word UNTIL is followed by a _____.

●●●

test-condition

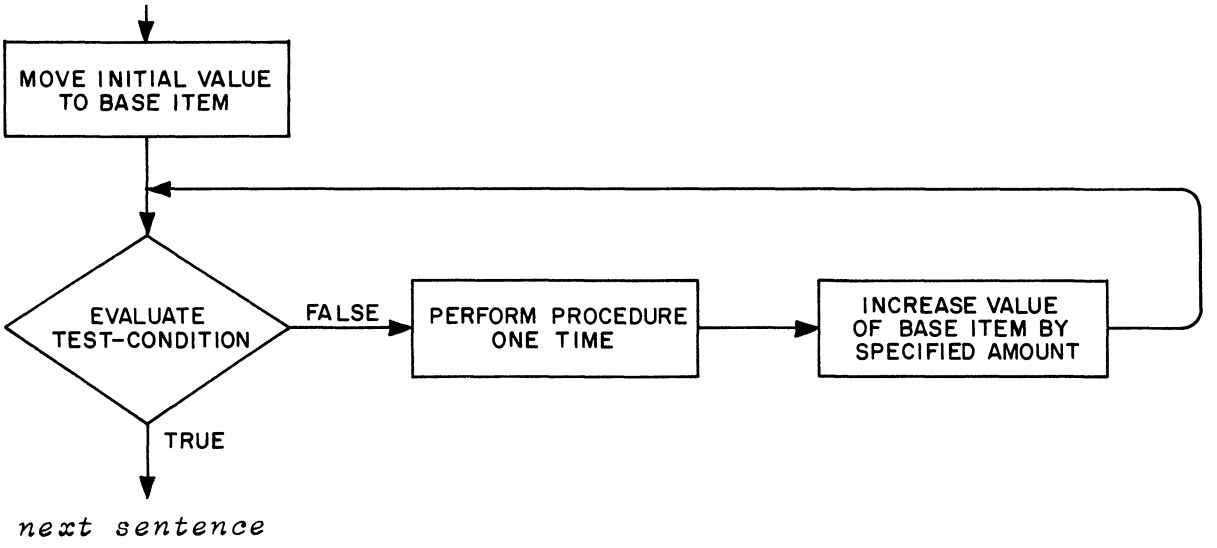
308 Here is the format of a PERFORM/VARYING statement. What should be written in place of each of the numbered boxes? (If you don't remember, look in the reference manual.)

PERFORM [1] VARYING [2] FROM [3] BY [4] UNTIL [5]

●●●

(1) the name of the procedure that is to be performed
(2) the name of the base item
(3) the initial value of the base item -- either the literal value, or the name of the data item that contains the value
(4) the amount by which the base item is to be increased each time the procedure is performed -- either the literal amount, or the name of data item that contains the amount
(5) a condition which is tested to determine when to stop performing the procedure

309 LOGIC DIAGRAM OF A "PERFORM/VARYING" STATEMENT



Execution of a PERFORM/VARYING statement begins with the setting of the initial value of the base item. Then the test-condition is evaluated before the procedure is performed at all. After each performance of the procedure, the base item is increased, and then the condition is tested again. As soon as the condition is found to be true, control passes to the next statement after the PERFORM statement.

●●●

310 This PERFORM statement will cause procedure Z to be executed

{ 9 times / 10 times / 11 times } .

```
    PERFORM Z, VARYING A FROM 1 BY 1,
        UNTIL A = 10.
```

●●●

9 times (The value of A will be increased to 10 right after the ninth performance of the procedure; immediately the condition of A will be tested and found to be "true" so there will not be a tenth performance.)

311 On a program sheet, rewrite the PERFORM statement in the previous frame so as to cause Z to be performed 10 times.

●●●

The best solution is to use a PERFORM/TIMES statement instead of a PERFORM/VARYING statement. It is a misuse of PERFORM/VARYING to use the base item merely as a counter.

```
PERFORM Z 10 TIMES.
```

If you chose to stick with PERFORM/VARYING, you might have corrected the statement in any one of these ways. All cause the same result.

```
PERFORM Z, VARYING A FROM 1 BY 1,
    UNTIL A = 11.
```

```
PERFORM Z, VARYING A FROM 1 BY 1,
    UNTIL A > 10.
```

```
PERFORM Z, VARYING A FROM 0 BY 1,
    UNTIL A = 10.
```

312 Why won't this statement function properly?

```
PERFORM B VARYING X FROM 1 BY 5
    UNTIL X = 45.
```

●●●

The value of X starts at 1 and is increased by 5 each time; therefore, it will never equal 45, and procedure B will be performed endlessly. *The error is fairly obvious in this entry because it uses X, or rather misuses X, as a counter, with literal values for the FROM, BY, and UNTIL amounts. The same error has a way of creeping in unnoticed when these amounts are represented by data names. Careful checking is always necessary.)*

LESSON 9

313 *We began Lesson 8 by saying that PERFORM statements were convenient and versatile -- but that their function could be accomplished by using other statements and other techniques which you had already studied. In a way, that's like saying that you don't have to use an automobile because a horse and buggy will get you there just as well; but recall what our point was -- that using PERFORM will result in a longer object program than using simpler statements to achieve the same end. The point becomes especially important when you are trying to squeeze a very large program into limited storage space.*

These same ideas apply to our next subject, which is another convenient and versatile feature of COBOL: "subscripting". Subscripting makes it easy to process data "tables" in storage. You will want to study the subscripting capability closely, because it enables you to handle your data in a rather different way. Instead of bringing it into storage piecemeal, you can bring large portions of data into storage at one time, and treat them as data tables. Or instead of calculating certain results for each transaction record, you might store a table of pre-calculated values and simply locate the appropriate value when it is needed. Thus, subscripting opens up a new dimension for solving problems with COBOL: the use of data tables.

●●●

314 Some definitions are in order. Here is what we mean by a "table" in COBOL:

(1) A table is a series of data items of the same type.

(2) All items in a table must have identical descriptions -- the same size, class, and usage.

(3) The items in a table must be adjacent to each other; that is, they must form a contiguous set of items in storage.

Keep in mind that an "item" is an area used to contain data -- or a "box" for data, if you will. Your mental picture of a table should be a string of boxes, all the same size, one box after another. To complete the picture, you can imagine that, in general,

{ the boxes will all contain the same data values
{ the boxes will contain different data values } .

●●●

the boxes will contain different data values

315 The definition of "table" also implies that the size of the items, the number of items, and the kind of data they contain are

{ determined by the programmer. / standardized and must be the same in all COBOL programs. }

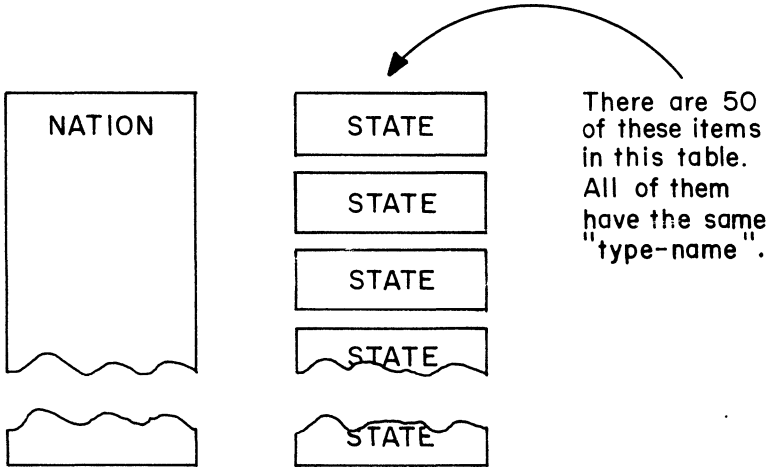
●●●

determined by the programmer

The rule stated that all items in a table must be the same size, but what the size is remains for the programmer to decide. Also, different tables may have different-size items and different numbers of items, and, of course, may contain different kinds of data.

316 The items in a table are not given unique, individual names. Instead, one name is assigned to the type of data item that is found in the table. In this book, we will refer to this name as the "type-name" of the items.

For example, in a table of the names of the states in the U. S., the items might be given the type-name "STATE". There are fifty states, so there would be fifty items in the table, each item called STATE. The structure of such a table can be shown in this way:



While each item in the table is called STATE, the table, as a whole, has another name, which is _____.

●●●

NATION

317 *It should be clear that using the type-name by itself does not allow us to refer to a specific item in the table. Thus, in the table of states, using the name STATE alone would leave the compiler scratching its head, wondering "Which STATE are they talking about -- I've got fifty of them!"*

Subscripting enables us to refer to specific items in a table. At this point, get an idea of what subscripts are, and how they are written.

Reading assignment: Subscripting

●●●

318 To refer to an individual item in a table, the type-name must be

{ preceded by / followed by } a subscript.

●●●

followed by

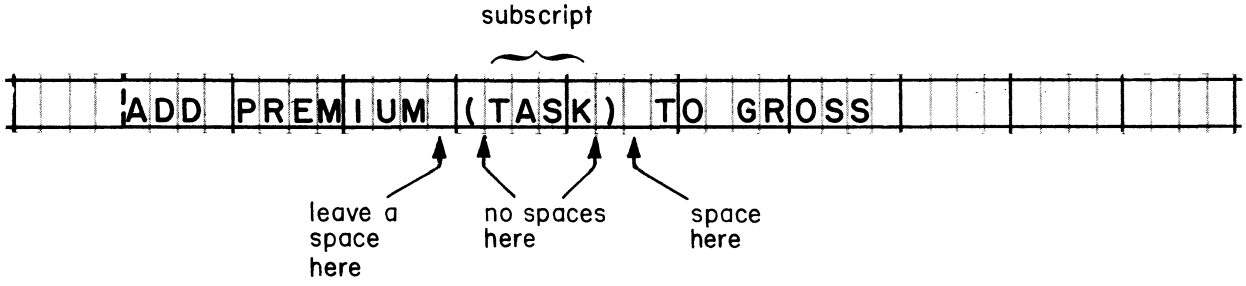
319 The subscript indicates where the item appears in the table (first, second, third, etc.). A subscript can be either a data name or a literal. Either way, the subscript must represent a positive whole number whose value is no smaller than 1 and no larger than the number of items in the table.

For the table of state names, the subscript could represent any number from _____ to _____.

●●●

1, 50

320 Subscripts are written after the type-name, and are enclosed in parentheses. Notice how this is done.



●●●

321 The example in the previous frame illustrates that subscripts are written in the [Procedure division] [Data division].

●●●

Procedure division ONLY

322 If the value of a subscript is 1, it stands for the first item in the table; if it is 2, it stands for the second item, and so on.

On a program sheet, use a literal subscript to move the name of the thirtieth state in our table of state names to an item called STATE-NAME.

●●●

```
MOVE STATE (30) TO STATE-NAME.
```

323 Most often, a data name is used as the subscript. A data-name subscript is used in the entry below. This entry will do exactly the same processing as the entry in the previous frame, when the value of STATE-NUMBER is 30.

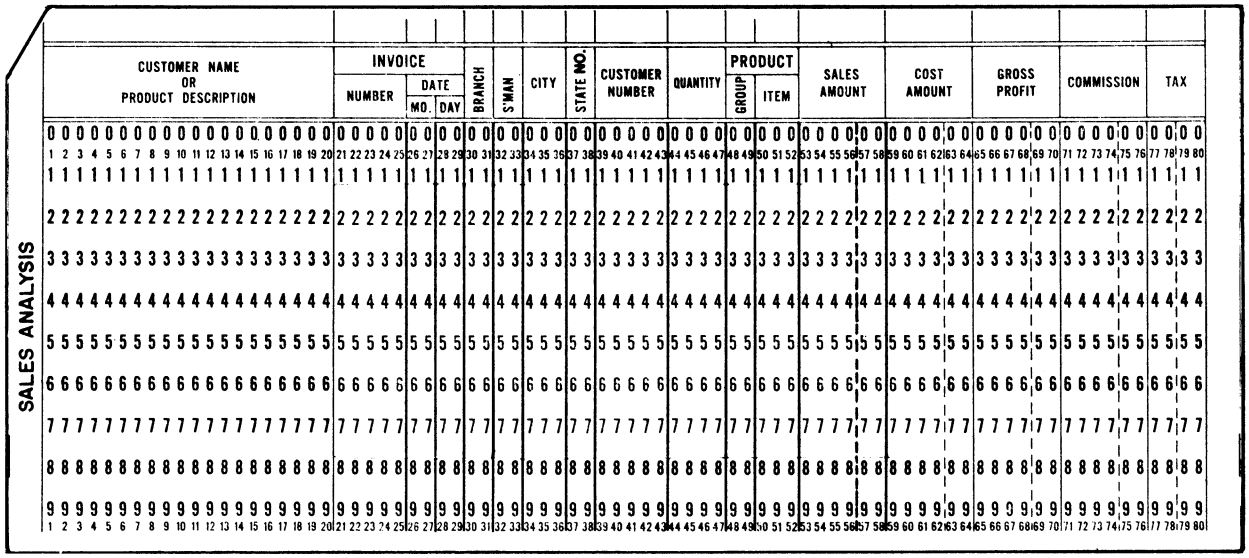
```
MOVE STATE (STATE-NUMBER) TO STATE-NAME.
```

The advantage of using a data name as a subscript is that the value represented by a data name is variable, whereas the value of a literal is _____.

●●●

constant

324 The value of a data-name subscript such as STATE-NUMBER might be determined by the contents of an input record. For instance, a state number is punched in columns 37-38 of the input card printed below.



The state numbers punched in this card can range from 01 to 50, with 01 being the code for Alabama, 02 for Alaska, and so on in alphabetical order, through 50 for Wyoming. The card contains only the number of the state; when we need the name of the state, we can move it from our table.

The state number from the card can serve as the subscript when referring to the table of state names,

{ provided that the table has been arranged in alphabetical order, so the number from the card corresponds to the place of that state in the table.
regardless of the arrangement of the table, since data can be located at random once it has been read into core storage. }

●●●

provided that the table has been arranged in alphabetical order, so the number from the card corresponds to the place of that state in the table.

325 Here is how subscripting works. When you use a data name as a subscript, the compiler generates a special subroutine in the object program. During the execution of the object program, that subroutine computes the storage address of the required item, using the current value of the subscript. The System/360 uses binary addresses, so the address computation is done most efficiently when the usage of the subscript is COMPUTATIONAL.

In our STATE-NUMBER example, the subscript comes from a card record, and so its usage is DISPLAY. You can guess that in cases like this, the special subroutine will also convert the data code of the subscript value to (what usage?) _____.

●●●

COMPUTATIONAL (binary)

326 When you use a literal as a subscript, the compiler develops the actual address of the table item during compilation. Therefore, the address { will / will not } be computed again during the execution of the object program.

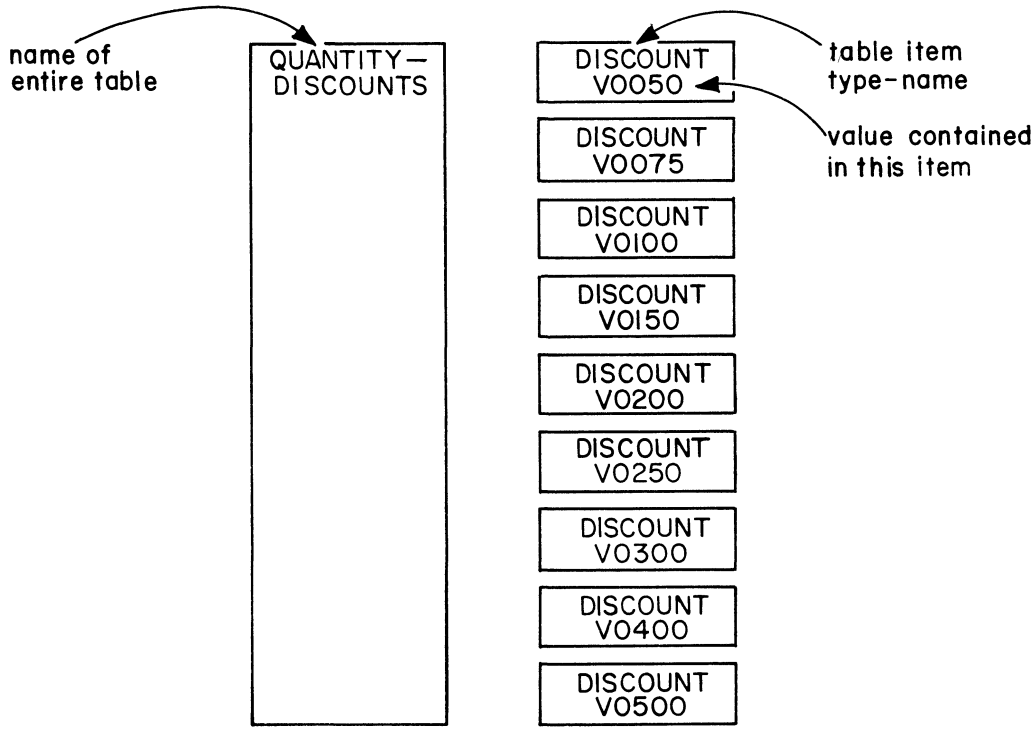
●●●

will not

327 *You have learned enough about tables and subscripting to be able to write procedural entries that contain subscripts to refer to items in tables. Here is a problem situation that will let you test your ability to write such entries.*

Suppose that a company gives its customers quantity discounts that vary from 1/2% to 5%, depending on the size of the order. There are nine discount percentages, and their values have been stored in a table which is diagrammed at the top of the next page.

Each customer order is assigned a rating number between 1 and 9. The rating number is the same as the place of the corresponding discount in the table. Thus, rating 1 is assigned to orders on which the discount is 1/2% (v0050); rating 3 corresponds to a discount of 1% (v0100); etc.



On a program sheet, write a procedural statement to calculate DISCOUNT-AMOUNT by multiplying ORDER-AMOUNT by the appropriate discount. (The rating number is in an item called RATING. The discount name is shown in the diagram above.)

●●●

```
COMPUTE DISCOUNT-AMOUNT =
    ORDER-AMOUNT * DISCOUNT (RATING).
```

You might also have written: MULTIPLY ORDER-AMOUNT BY DISCOUNT (RATING) GIVING DISCOUNT-AMOUNT.

328 When the table diagrammed in the preceding frame is put into storage, the items will actually contain [1] [2] [3] [4] digits.

●●●

4 digits ONLY *(All items in a table must be the same size, so it is not possible to store v0075 for one item, and v05 for another item.)*

329 True or false:

In the discount problem, the computer is able to locate the discount that corresponds to a rating number because the numbers 1 through 9 are actually stored in the table along with the discount amounts.

●●●

False. *The computer does not scan through the table looking for a discount that is identified by a particular number. Instead, it uses the subscript to compute the exact location of the desired discount, and goes directly there to obtain the data. Each item in the QUANTITY-DISCOUNT table contains only four digits as shown in the diagram.*

330 *Now that you have seen how to use subscripts to refer to an item in a table, let's talk about how to set up tables. You are permitted to use subscripts after a data name only if its item description entry contains an OCCURS clause.*

Reading assignment: OCCURS clause (Option 1)

●●●

331 These two entries are all that is required to set up the QUANTITY-DISCOUNTS table we used in the discount problem. Note that the level-02 entry, which contains the OCCURS clause, defines a string of nine identical items.

```
01  QUANTITY-DISCOUNTS, COMPUTATIONAL-3.
    02  DISCOUNT, OCCURS 9 TIMES,
        PICTURE SV999.
```

Do these entries also cause the discount values to be put into the table items?

●●●

No. *Setting up a table and filling it with data are two separate considerations. A little later we will discuss how to get data values into the table.*

332 You might have no need to refer to the discount table as a whole, using the name QUANTITY-DISCOUNTS. Still it is not correct to define the table as shown below. What rule does this entry violate?

```
01  DISCOUNT, OCCURS [illegible] TIMES.
        PICTURE SV9[illegible] COMPUTATIONAL-3.
```

●●●

An OCCURS clause must not be used in a level-01 entry.

333 On a program sheet, write the entries that define a table called NATION which will contain the names of the fifty states. Allow fourteen positions for each STATE, to accommodate the longest state names (NORTH CAROLINA and SOUTH CAROLINA), even though this means there will be ten blanks in the items that contain the shortest state names (IOWA, OHIO, and UTAH).

●●●

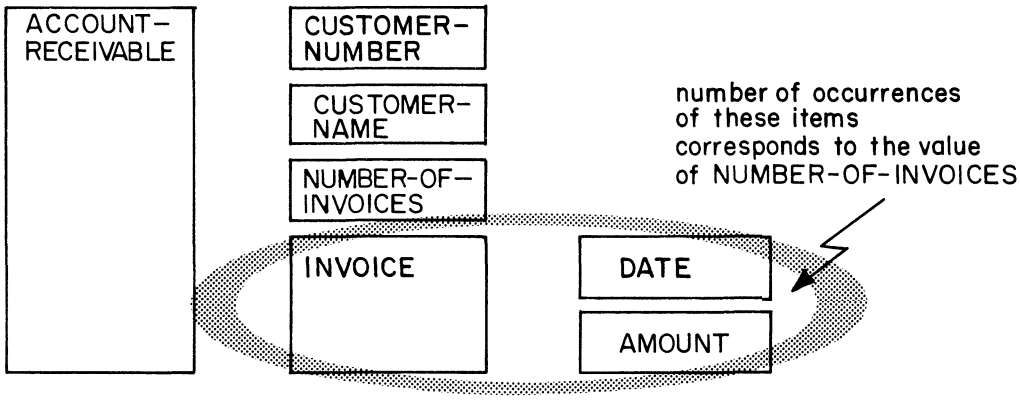
```
01  NATION.
    02  STATE, OCCURS 50 TIMES,
            PICTURE A(14).
```

334 *In the table of discounts and in the table of states, we knew exactly how many items there would be. This enabled us to use Option 1 of the OCCURS clause. However, in some cases, the length of a table is variable. For example, part of an input record might be a table with one item for each transaction; if there had been five transactions, the table in that record would contain five items, but there might be 250 items in the table in another record if there were that many transactions. Such variable length tables can be handled in COBOL, provided that there is a data item which specifies how many items are in the table each time. The name of this data item must be written in an "OCCURS/DEPENDING ON" clause.*

Reading assignment: OCCURS clause (Option 2)

●●●

335 Study this example of a variable length record and its record description. The record can contain information about several invoices. Since there may be a number of consecutive appearances of the INVOICE item, we can properly say that there is a table within the record. Also, because INVOICE has been described using an OCCURS clause, all references to INVOICE in the Procedure division must be subscripted.




```
01  ACCOUNT-RECEIVABLE.
    02  CUSTOMER-NUMBER, PICTURE 9(10).
    02  CUSTOMER-NAME, PICTURE A(25).
    02  NUMBER-OF-INVOICES, PICTURE 99.
    02  INVOICE, OCCURS 25 TIMES,
            DEPENDING ON NUMBER-OF-INVOICES.
        03  DATE, PICTURE 9(6).
        03  AMOUNT, PICTURE 9(5)V99.
```

According to the record description, INVOICE may appear

{ no fewer than 25 times
no more than 25 times
as many as 99 times } in each record.

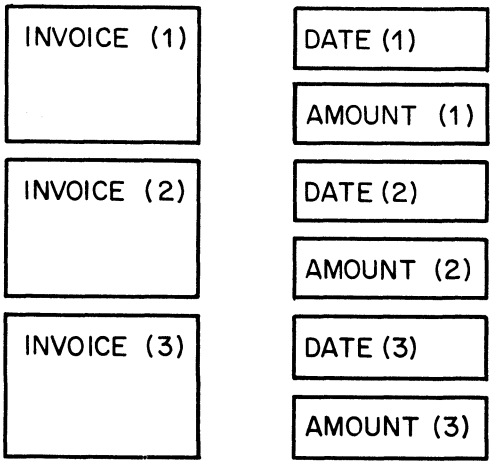
●●●

no more than 25 times

The integer written in an OCCURS/DEPENDING ON clause specifies the maximum number of occurrences. Even though the picture of NUMBER-OF-INVOICES is 99, the value of this item will not exceed 25.

336 In the record illustrated in the previous frame, INVOICE is subdivided into DATE and AMOUNT. These items are part of the table, and occur just as often as INVOICE does. The names "DATE" and "AMOUNT" are therefore type-names just as "INVOICE" is a type-name, and none of these names may be written in a procedure unless it is subscripted.

The subscript of the group item, INVOICE, applies to the items in the group, DATE and AMOUNT. This is shown in the diagram below, in which the table contains three INVOICEs and literal subscripts are included after each item name.



Write an entry which subtracts PAYMENT from the AMOUNT of the second INVOICE.

●●●

```
SUBTRACT PAYMENT FROM AMOUNT (2).
```

337 *In practice, we would not want to use literal subscripts to process the invoice data. After all, there may be as many as 25 invoices in a record, and using literal subscripts would involve writing 25 sets of processing statements. In place of this, we will define a working-storage item to serve as the subscript. We will set the value of this item at 1 to process the first invoice data; then add 1 to the item to repeat the process for the second invoice; and so on, until its value is equal to the number of invoices in the record. The program excerpts below show how this might be done. Study them carefully.*

```
DATA DIVISION.

WORKING-STORAGE SECTION.

77  POINTER, PICTURE S99, COMPUTATIONAL.

PROCEDURE DIVISION.

A.   READ ACCOUNTS-RECEIVABLE-FILE;
         AT END, GO TO COMPLETION.
     MOVE 1 TO POINTER.
B.   MOVE DATE (POINTER) TO PURCHASE-DATE.
     MOVE AMOUNT (POINTER) TO AMOUNT-DUE.
     PERFORM BILLING-ROUTINE.
C.   IF POINTER = NUMBER-OF-INVOICES,
         GO TO BILL-OUTPUT;
     OTHERWISE, ADD 1 TO POINTER, GO TO B.
```

●●●

The objective of this process is to read a record and then to execute a loop (procedure B) as many times as there are invoices in the record. The loop might also have been controlled by a PERFORM/VARYING statement: PERFORM B, VARYING POINTER FROM 1 BY 1, UNTIL POINTER > NUMBER-OF-INVOICES.

338 Each time a data-name subscript is used in the Procedure division, an address calculation will be called for in the object program. So, in the example on the preceding page, address calculation will be done twice, once for DATE (POINTER), and again for AMOUNT (POINTER).

Naturally, we want to keep the number of address calculations to an absolute minimum. Usually, we can limit the number to one when we are processing a table. We can do this by using a technique that we have used before to eliminate other inefficiencies, such as extra data code conversions -- we will define a work area which can contain one entire table item. The processing of the data would actually be done from the work area, not from the table.

The work area for our invoice problem could be defined in this way:

```
01  INVOICE-DATA.
    02  PURCHASE-DATE, PICTURE 9(6).
    02  AMOUNT-DUE, PICTURE 9(5)V99.
```

Once the data for one invoice has been moved to the work area, subscripting is no longer needed to process the data. A subscript is used, in this case, only to locate the desired invoice data in the table and bring that data out to a work area. (There is an even greater advantage when the table item is subdivided into more than two smaller items. You can imagine a table item being subdivided into a dozen items; certainly in a case like that it would be much more efficient to use subscripting once to move the item to a work area, than to use subscripting a dozen times to process the data directly from the table.)

On a program sheet, write the statement tht would be necessary to move the data for an INVOICE to the work area, INVOICE-DATA. The subscript item, POINTER, would still be defined and progressively increased as shown in the previous frame.

●●●

```
    MOVE INVOICE (POINTER) TO INVOICE-DATA.
```

LESSON 10

339 *So far, we have briefly discussed: (1) what a table is; (2) what subscripts are and how they are written; (3) how tables are defined in the Data division; and (4) how they may be processed. Now, we will turn our attention to the question of getting values into a table. There are two general ways of doing this: first, by defining the table in the File section and reading input data into it; second, by defining the table in the Working-Storage section, and writing constant values for it in the program.*

The first way is definitely required when the table is part of the input record. This was true of the ACCOUNT-RECEIVABLE record which you just studied; when an input record was read, the values of the invoice dates and amounts were put into the table. There are several other reasons for treating table data as an input file. The data might be very long, and brought into storage a segment at a time. It might be used by more than one program, and read in from a direct access device or from magnetic tape as it was needed. Or the data might change frequently, and be kept in a file of punched cards to make updating easier.

The second way -- writing the table values in the program itself -- is normally used when the table is not too long, and when its values do not change very often.

●●●

340 Suppose it has been decided to store the table data as one long record on a direct access device. To get that data into the computer, we will have to [open the file] [read the file] [close the file].

●●●

ALL of these

341 The file is called TABLE-FILE. Here are the entries that open and read the file. How many times will the read statement have to be executed in order to bring all of the table data into storage?

```
1.  OPEN INPUT TABLE-FILE.
    READ TABLE-FILE; AT END, GO TO 2.
```

●●●

Only once. The data has been stored as one long record, so a single read will bring all of it into storage.

342 On a program sheet, write the entries that are needed to define the table. The table as a whole is called TABLE; it consists of 250 items whose type-name is TABLE-ELEMENT. Each item in the table is seven digits long, including two decimal places, and is stored in packed-decimal form.

●●●

```
01  TABLE.
    02  TABLE-ELEMENT, OCCURS 250 TIMES,
        PICTURE S9(5)V99, COMPUTATIONAL-3.
```

343 *It may be helpful to see how these entries fit in with other program entries, such as the SELECT entry in the Environment division and the file description in the Data division. Keep in mind that these program excerpts are only concerned with getting the table data into the computer; other entries would be needed to process the data.*

Environment division, Input-Output section, File-Control paragraph:

```
    SELECT TABLE-FILE, ASSIGN TO 'TABLE'
        UTILITY 2311 UNIT.
```

Data division, File section:

```
FD  TABLE-FILE, RECORDING MODE IS V,
    LABEL RECORDS ARE STANDARD,
    DATA RECORD IS TABLE.

01  TABLE.
    02  TABLE-ELEMENT, OCCURS 250 TIMES,
        PICTURE S9(5)V99, COMPUTATIONAL-3.
```

Procedure division:

```
1.  OPEN INPUT TABLE-FILE.
    READ TABLE-FILE; AT END, GO TO 2.
2.  CLOSE TABLE-FILE.
```

344 If the table data file contains more than one record, each record must be read in turn into an input area, and then moved into its proper place in the table. Suppose that the data for a table is punched into 100 cards. The 80 characters punched in each card are the data for one item in the table. The input file, input record, and table have been defined in this way:

```
DATA DIVISION.
FILE SECTION.
FD  TABLE-FILE, RECORDING MODE F, LABEL
    RECORDS OMITTED, DATA RECORD TABLE-INPUT.
01  TABLE-INPUT, PICTURE X(80).

WORKING-STORAGE SECTION.
01  TABLE.
    02  TABLE-ELEMENT, OCCURS 100 TIMES,
        PICTURE X(80).
```

The problem now is to read the card records, and to put the data from the first card into the first TABLE-ELEMENT, the data from the second card into the second TABLE-ELEMENT, and so on. See if you can solve this problem. Write the procedural entries to get the data from the TABLE-FILE into the TABLE; you may also need to write an additional data description entry in order to accomplish this.

●●●

Probably the "natural" solution to this problem is to use subscripting when moving the data into the table. I have defined an additional working-storage item to serve as the subscript:

```
77  NUMBER, PICTURE S999, COMPUTATIONAL.
```

Then I have increased the value of the item by 1 each time an input record is moved. In this way, the first record is moved to TABLE-ELEMENT (1), the second record to TABLE-ELEMENT (2), and so on:

```
1.  OPEN INPUT TABLE-FILE. MOVE 1 TO NUMBER.
2.  READ TABLE-FILE; AT END, GO TO 3.
    MOVE TABLE-INPUT TO
        TABLE-ELEMENT (NUMBER).
    ADD 1 TO NUMBER. GO TO 2.
3.  CLOSE TABLE-FILE.
```

345 *When a table is small and its values do not change very often, it can be written in the Working-Storage section of the COBOL program and constant values can be defined for it. The table, with the values already in it, is then an integral part of the object program, and is loaded into storage along with the computer instructions. (To change these values, you would merely alter the source program, and compile a new object program.)*

In order to set up this kind of table, you must write two record descriptions:

(1) First, describe a record and supply values for it.

(2) Then, describe a second record which redefines the first one; in this record description, name the table item and specify how many times it occurs.

This method must be followed because the rules of COBOL forbid a VALUE clause to appear in an entry that contains an OCCURS clause. In fact, you aren't allowed to write a VALUE clause in any entry that is subordinate to an entry that contains an OCCURS clause, either. So we must keep these clauses in separate record descriptions, and by using a REDEFINES clause in the second record description, we will specify that the two descriptions apply to the same area of storage.

Although you have used both the VALUE clause and the REDEFINES clause before, be sure to read what the reference manual has to say about them.

Reading assignment: VALUE clause
REDEFINES clause

●●●

346 *All of the "musts" and "may nots" in the reference manual can become somewhat confusing, but the way in which we will use these clauses is really quite simple and straightforward. After we have written the entries to set up a table in working storage, you may want to re-read the manual's rules to assure yourself that our method is perfectly legal.*

This is the problem we will work on. We want a table of the names of the days of the week: SUNDAY, MONDAY, ... SATURDAY. The table item type-name is to be DAY, so that DAY (1) will refer to the value 'SUNDAY', DAY (2) to 'MONDAY', etc.

But remember the approach that was described above. The first thing that we will do is not to describe this table, but rather to write an ordinary record description in which we will give the values of the data items.

On a program sheet, write a description of a record named DAYS-OF-THE-WEEK. Define seven items within the record. We will not need any names for these items, so call each of them FILLER. Make each item nine characters long, and make their values the names of the days of the week beginning with Sunday.

●●●

```
WORKING-STORAGE SECTION.

01  DAYS-OF-THE-WEEK.
    02  FILLER, PICTURE A(9), VALUE 'SUNDAY   '.
    02  FILLER, PICTURE A(9), VALUE 'MONDAY   '.
    02  FILLER, PICTURE A(9), VALUE 'TUESDAY  '.
    02  FILLER, PICTURE A(9), VALUE 'WEDNESDAY'.
    02  FILLER, PICTURE A(9), VALUE 'THURSDAY '.
    02  FILLER, PICTURE A(9), VALUE 'FRIDAY   '.
    02  FILLER, PICTURE A(9), VALUE 'SATURDAY '.
```

347 Immediately below the record description you have just written, write one that redefines DAYS-OF-THE-WEEK. Name it WEEK, and indicate that it is composed of the item DAY, which occurs seven times. DAY must be nine characters long.

●●●

```
01  WEEK, REDEFINES DAYS-OF-THE-WEEK.
    02  DAY, OCCURS 7 TIMES, PICTURE A(9).
```

348 *The two record descriptions you have just written together give us the table we wanted. Take a moment to look back over your work to make sure you know how the record descriptions work together.*

●●●

349 Now use the same method to set up a table in working storage that contains the names of the twelve months, January through December.

Call the first record "MONTH-NAMES" and the second record "TABLE-OF-MONTHS". Make the table item type-name "MONTH".

●●●

```
01  MONTH-NAMES.
    02  FILLER, PICTURE A(9), VALUE 'JANUARY  '.
    02  FILLER, PICTURE A(9), VALUE 'FEBRUARY '.
    02  FILLER, PICTURE A(9), VALUE 'MARCH    '.
    02  FILLER, PICTURE A(9), VALUE 'APRIL    '.
    02  FILLER, PICTURE A(9), VALUE 'MAY      '.
    02  FILLER, PICTURE A(9), VALUE 'JUNE     '.
    02  FILLER, PICTURE A(9), VALUE 'JULY     '.
    02  FILLER, PICTURE A(9), VALUE 'AUGUST   '.
    02  FILLER, PICTURE A(9), VALUE 'SEPTEMBER'.
    02  FILLER, PICTURE A(9), VALUE 'OCTOBER  '.
    02  FILLER, PICTURE A(9), VALUE 'NOVEMBER '.
    02  FILLER, PICTURE A(9), VALUE 'DECEMBER '.
01  TABLE-OF-MONTHS REDEFINES MONTH-NAMES.
    02  MONTH, OCCURS 12 TIMES, PICTURE A(9).
```

By coincidence, the lengths of the longest month name and the longest day name happen to be the same, so the pictures of the table items in this table and the last one are both A(9).

350 *Our discussion of getting values into tables rounds out the subject. You have now seen how to define a table, how to fill it with values, and how to use subscripts to process it.*

Needless to say, the subject has many other aspects. In fact, an entire book might be devoted to tables and subscripting. Very briefly, we will look at two of these aspects: first, how to do table "look up", and second, how to define tables within tables.

●●●

351 Table "look up" is done by applying rules that you already know about tables. The need to look up information in a table arises when we don't know exactly where the desired data appears -- that is, when we don't know whether it is in the first item of the table, in the second, third, etc. The fact is that there are many cases in which an item's place in a table does not serve to identify it; in such cases, we have to examine the value of the data in an item in order to find out whether this is the item we want.

To get the idea, suppose we have a table of employees' hourly wage rates, some of whose values might look like this:

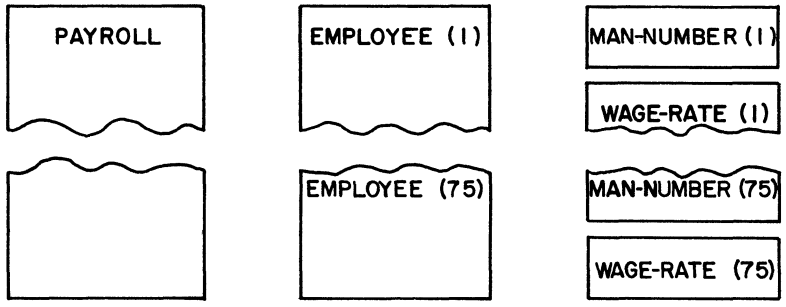
MAN-NUMBER	WAGE-RATE
26725	2v250
28224	3v125
30096	1v750
30105	1v275
36259	2v500

As the payroll is prepared, we want to look up the wage rate that corresponds to each man number. For instance, when the time card for man number 30096 comes up, we want to find the table item for 30096 and get the man's wage rate: $1.75 an hour. The table changes from time to time as employees are added and removed, so we cannot know what position man 30096 will occupy in the table.

The solution to this problem is to go through the table, item by item, comparing man numbers until we find an equal comparison. To step our way through the table, we can use a subscript item; set its value at 1 to begin with, and use it to compare man numbers; increase its value progressively and make repeated comparisons until an equal condition occurs.

●●●

352 Here is one way in which the table of wage rates can be structured. The man number and wage rate for each employee are stored together in the table. In this example, there are 75 employees on the payroll.



Which set of entries below corresponds to the record structure diagrammed above?

(1)

```
01  PAYROLL.
    02  EMPLOYEE.
        03  MAN-NUMBER OCCURS 75 TIMES,
            PICTURE 9(5).
        03  WAGE-RATE OCCURS 75 TIMES,
            PICTURE 9(5).
```

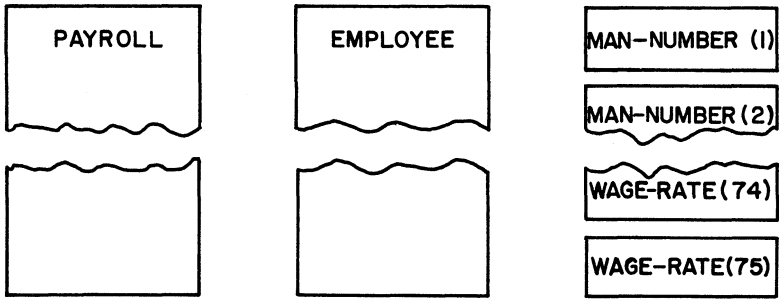
(2)

```
01  PAYROLL.
    02  EMPLOYEE OCCURS 75 TIMES.
        03  MAN-NUMBER, PICTURE 9(5).
        03  WAGE-RATE, PICTURE 9V999.
```

●●●

(2)

Record description (1) defines <u>two</u> tables, one containing a string of 75 man numbers, the other containing a string of 75 wage rates. The structure of that record would be:

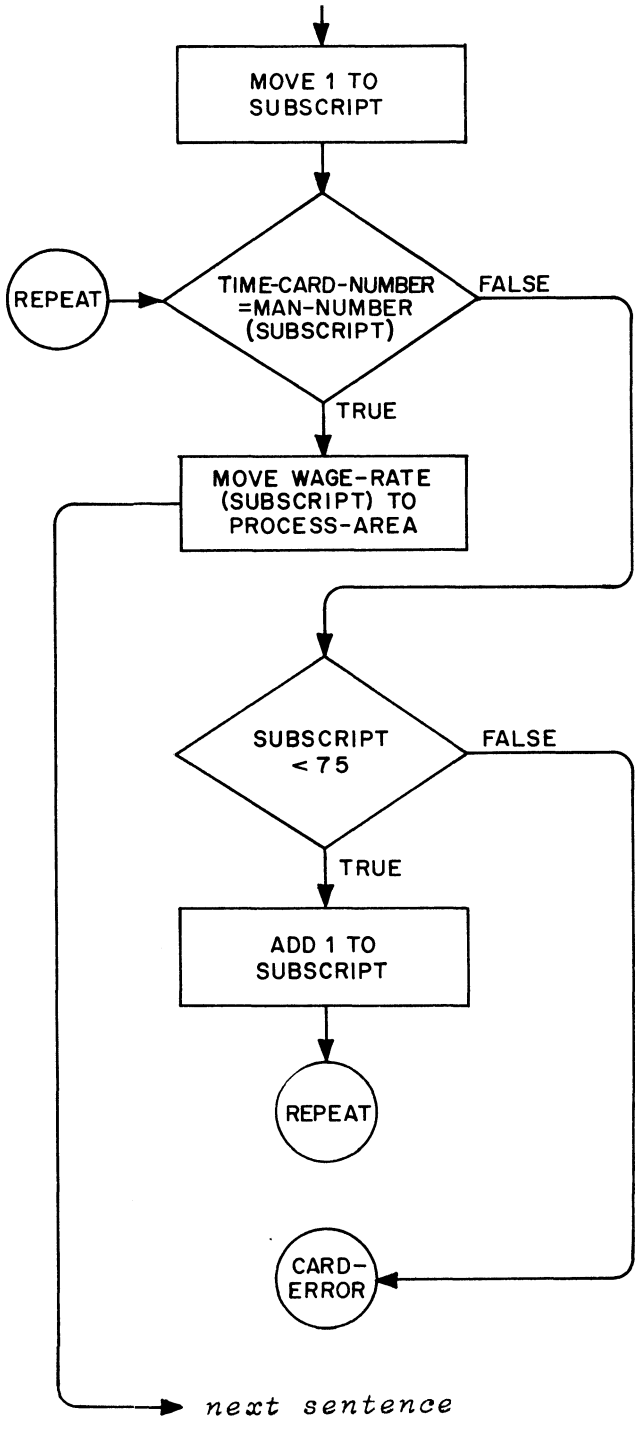


353 Now that we have defined the payroll table, let's take it for granted that the table would be filled with data by one of the methods that you have studied. Our next problem is to program a table look-up which will look at one man number after another until an equal comparison is found.

The flowchart excerpt on the right shows one way in which the look-up can be done. Just prior to this excerpt, the program has called for a time card to be read. Then a 1 is moved into a working storage item called SUBSCRIPT; this puts us at the beginning of the table as we make our first comparison of a MAN-NUMBER from the table with the TIME-CARD-NUMBER. If the comparison is equal we move the corresponding WAGE-RATE to a PROCESS-AREA. Otherwise, the subscript is increased by one, and control branches back to the comparison point.

Notice that this process checks to make sure the subscript does not exceed 75, which is the size of the table. If we have searched all 75 table items without finding an equal comparison, control branches to a CARD-ERROR procedure.

Write the procedural entries that correspond to the steps in this flowchart.



●●●

The solution for this frame is printed at the top of the next page.

```
    MOVE 1 TO SUBSCRIPT.
REPEAT.
    IF TIME-CARD-NUMBER = MAN-NUMBER
    (SUBSCRIPT), MOVE WAGE-RATE (SUBSCRIPT)
    TO PROCESS-AREA;
    OTHERWISE, IF SUBSCRIPT < 75,
    ADD 1 TO SUBSCRIPT, GO TO REPEAT;
    OTHERWISE, GO TO CARD-ERROR.
```

354 *We could devote a lot more time to exploring various ways of doing table look up in COBOL, but I think that one exercise has been enough to give you a general idea of how to go about it. Similarly, we will take only a short look at our next topic: tables within tables.*

In COBOL, it is possible to have another table within each table item; in fact, you can have still another table within that table. This enables us to process tables that have two or three "dimensions", that is, tables in which the data is broken down two or three ways.

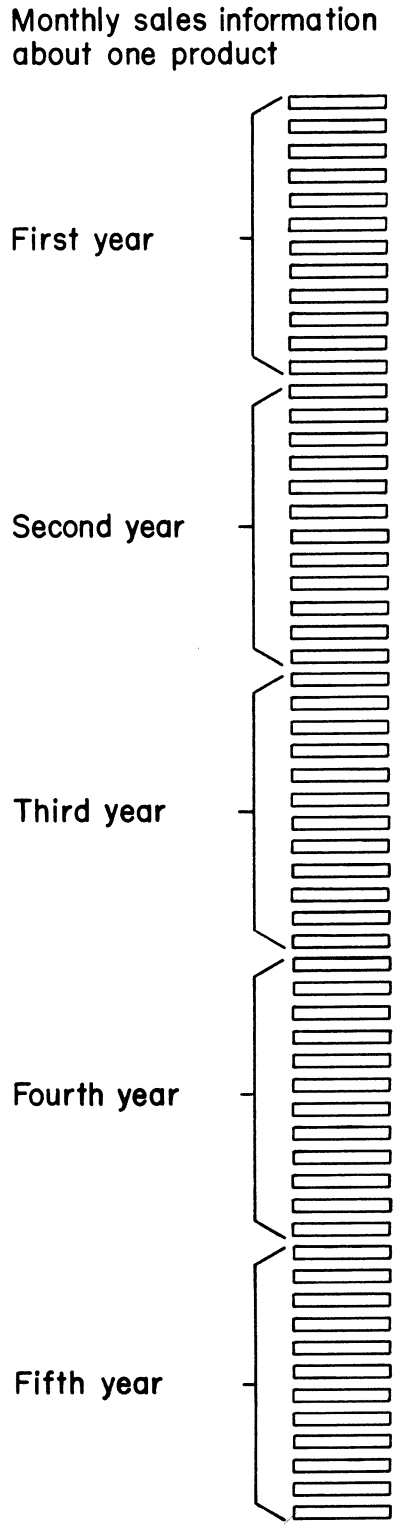
Re-read the reference manual information on Subscripting, this time paying particular attention to what is said about more than one level of subscripting.

Reading assignment: Subscripting

●●●

355 *We will study one example of a "multi-level" table, to see how it is defined and how "multi-level" subscripts are written.*

Suppose that your company wants to have a table of monthly sales figures for the last five years, for twenty different products. This really amounts to a string of 1,200 monthly sales totals grouped into 20 product categories, then divided into 5 yearly groups, and arranged in order by months within each year. The diagram at the right may help you to visualize what the string of items for one product will look like.



From the diagram, you can conclude that this particular table

{ does / does not }

contain a thirteenth item for each year which contains the yearly sales total.

●●●

does not

356 Each yearly group is simply made up of (how many?) _____ monthly-sales items.

●●●

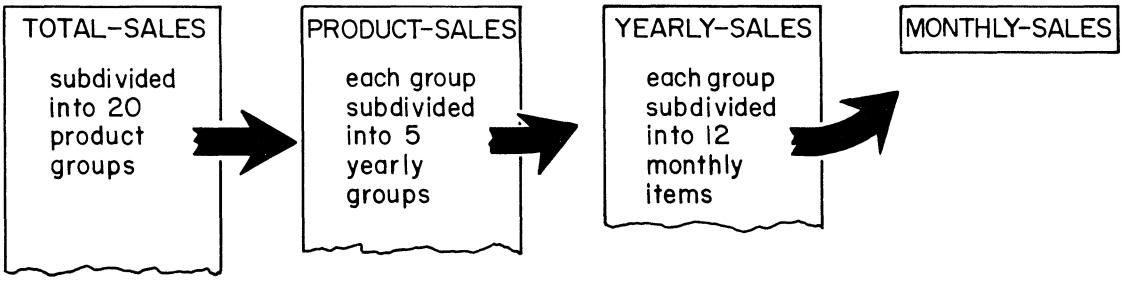
12

357 There are (how many?) _____ monthly-sales items for each product.

●●●

60

358 In COBOL, our thinking must proceed from the largest to the smallest item; in planning our description of this record, therefore, we must begin with the record as a whole. This drawing suggests how we may think of the structure of this record and its progressive subdivision.



Here is the start of the record description. See if you can complete it. The elementary item, MONTHLY-SALES, consists of a sign plus seven digits, including two decimal places, and is stored in packed-decimal format.

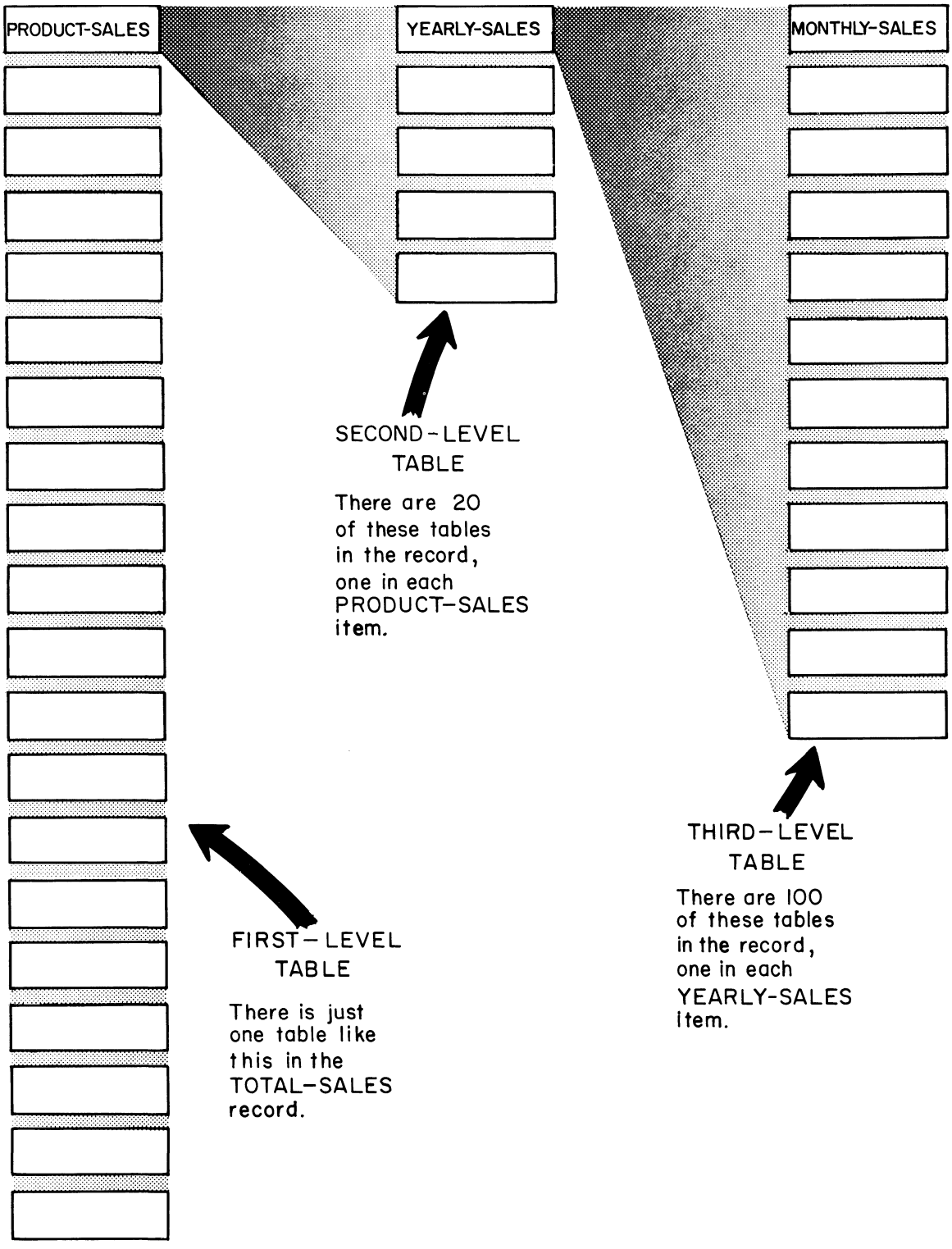
```
01  TOTAL-SALES.
    02  PRODUCT-SALES, OCCURS 20 TIMES.
```

●●●

```
01  TOTAL-SALES.
    02  PRODUCT-SALES, OCCURS 20 TIMES.
        03  YEARLY-SALES, OCCURS 5 TIMES.
            04  MONTHLY-SALES,
                OCCURS 12 TIMES,
                PICTURE S9(5)V99,
                COMPUTATIONAL-3.
```

359 *By having three OCCURS clauses in one record description, we have created a three-level table. (Three levels of tables are the maximum permitted in one record.) Each item of the first table contains a second table, and each item of the second table contains a third table. The illustration on the opposite page shows how the total number of tables is multiplied by having tables within tables. Study the illustration for a moment before proceeding to the next frame.*

●●●


PRODUCT-SALES
YEARLY-SALES
MONTHLY-SALES
SECOND-LEVEL TABLE
There are 20 of these tables in the record, one in each PRODUCT-SALES item.
THIRD-LEVEL TABLE
There are 100 of these tables in the record, one in each YEARLY-SALES item.
FIRST-LEVEL TABLE
There is just one table like this in the TOTAL-SALES record.

360 As in the tables you studied before, subscripts are required when you refer to items in a multi-level table. The problem of referring to an individual item in one of the tables is complicated a bit by the fact that there is actually a large number of tables in this record.

For example, the computer would be confused if you referred to MONTHLY-SALES (7) or YEARLY-SALES (2), although it would have no trouble finding PRODUCT-SALES (10). Can you explain why?

●●●

If you wrote MONTHLY-SALES (7), presumably you want the sales amount for the seventh month in one of the monthly sales tables -- but which table? We have seen that there are really a hundred of them. By the same token, it doesn't make sense to call for the second year's figures -- the second year for which product? On the other hand, there is only one product sales table, so PRODUCT-SALES (10) leads the computer directly to the tenth PRODUCT-SALES group.

361 Incidentally, if you wrote "MOVE PRODUCT-SALES (10) TO WORK-AREA", you would move a string of (how many?) _____ monthly sales amounts.

●●●

60

362 One subscript is sufficient to refer to an item in the first-level table. But you must use two subscripts after the name of an item in the second-level table, and three subscripts for an item in the third-level table.

When two or three subscripts are written after a name, they are all written inside a single pair of parentheses and they are separated by commas.

Which is correct:

{ MONTHLY-SALES (PRODUCT), (YEAR), (MONTH) }
{ MONTHLY-SALES (PRODUCT, YEAR, MONTH) }
{ (MONTHLY-SALES, PRODUCT, YEAR, MONTH) }

●●●

MONTHLY-SALES (PRODUCT, YEAR, MONTH)

363 Multiple subscripts are written in the same order as the table levels were defined in the record description. Thus, the first subscript tells where the item may be found in the first-level table; the second subscript tells the item's location in the second-level table; and the third subscript tells its location in the third-level table.

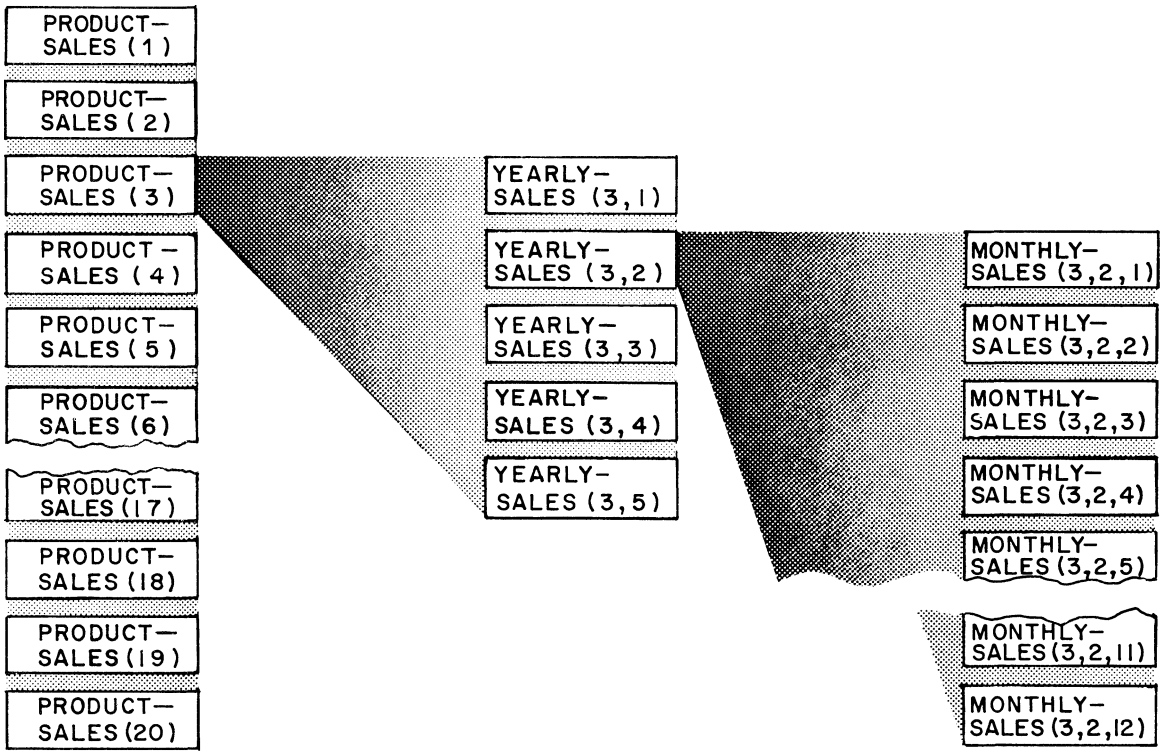
To address a YEARLY-SALES item in our sample table, the first subscript must indicate which product, and the second subscript must indicate which year you want. Therefore, YEARLY-SALES (3, 5) stands for

{ the third year's sales amounts for the fifth product. }
{ the sales amounts for the third product in the fifth year. }

●●●

the sales amounts for the third product in the fifth year

364 This drawing shows how certain items in our record would be referred to in procedures, if literal subscripts were used. The subscripting scheme is not hard to follow; you can think of it as adding another subscript for each additional table level. So (3) is the subscript for the third product; (3, 2) are the subscripts for the third product, second year; and (3, 2, 4) are the subscripts for the third product, second year, fourth month.



You could get the sales amount for the seventh product, fourth year, tenth month by addressing [PRODUCT-SALES (7, 4, 10)] [YEARLY-SALES (7, 4, 10)] [MONTHLY-SALES (7, 4, 10)].

●●●

MONTHLY-SALES (7, 4, 10). *You must use the type-name which has been defined for the table level from which you want an item.*

365 In the event that you needed to process every monthly sales item in the table, you would define three subscript items in working storage, possibly calling them PRODUCT, YEAR, and MONTH. You would then refer to the table items as MONTHLY-SALES (PRODUCT, YEAR, MONTH). To begin with, you would make all of the subscripts equal to 1. Keeping the values of PRODUCT and YEAR at 1, you would progressively increase the value of MONTH by 1 until it reached 12.

At this point, you would keep the value of PRODUCT at 1, but change the value of YEAR to _____, and change the value of MONTH to _____.

●●●

2; 1

366 After you had processed the amounts for every month of the five years for the first product, you would change the value of PRODUCT to _____, YEAR to _____, and MONTH to _____.

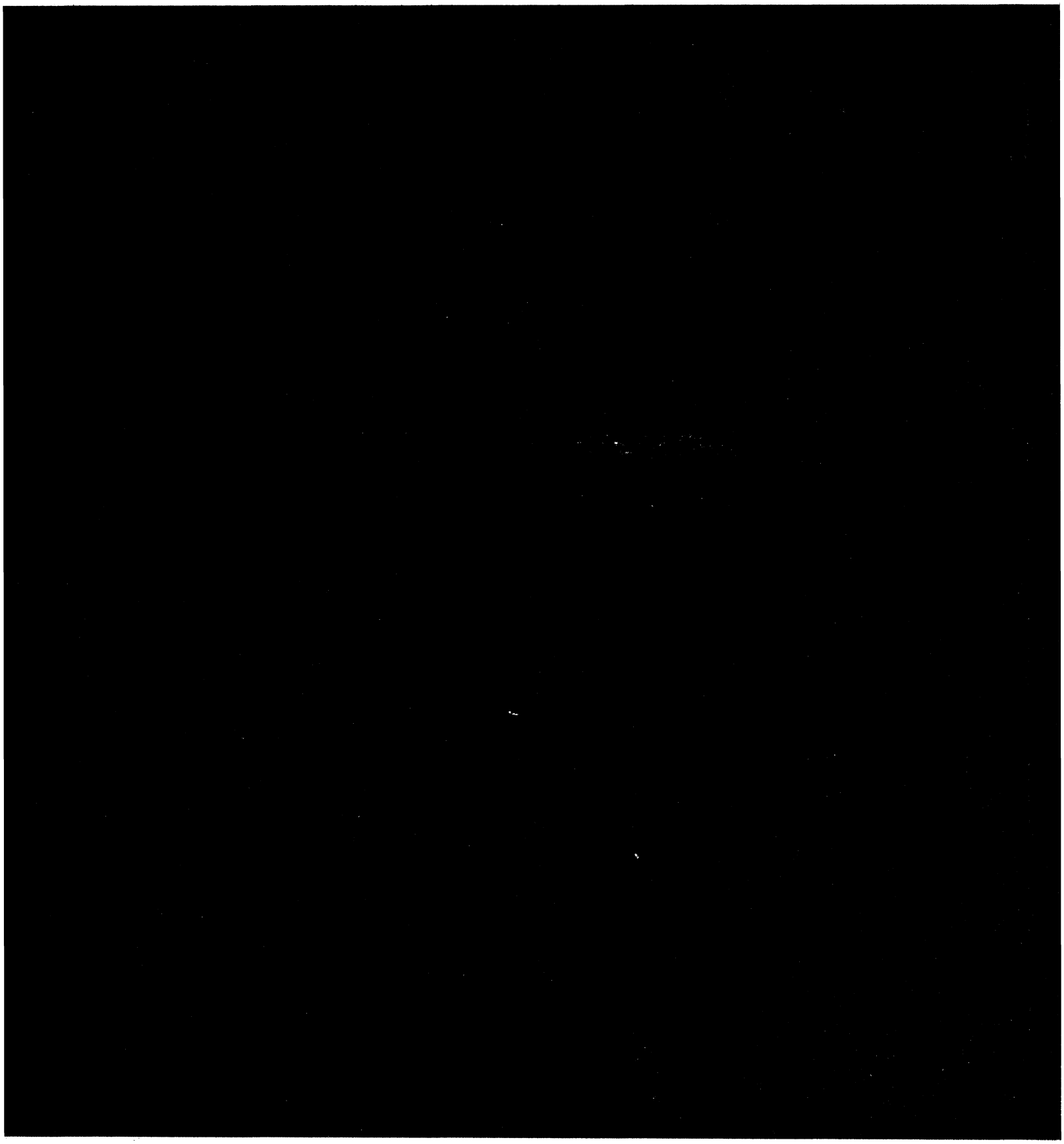
●●●

2; 1; 1

367 *We needn't belabor the point. You can see that the processing of multi-level tables is a fairly straightforward application of the fundamental operations that you studied in the earlier lessons of this book -- moving data, adding numbers, and making decisions. For that matter, you have seen that defining and addressing multi-level tables does not involve any new COBOL words; this function is just another application of entries that you learned about a long time ago.*

I am quite certain that most of your future work in COBOL will consist of applying familiar principles to new situations. In this series of short courses on COBOL, you have studied nearly all of the features of the language. Most important, you have learned the fundamentals which will make it easy for you to understand and apply the additional details which you may encounter in reference manuals.

●●●



International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York