4. COBOL PROGRAMMING

4.1 Introduction to COBOL

4.2 COBOL Program Outline

4.3 COBOL Numeric Formats

4.4 COBOL Numeric-Edited Output

4.5 COBOL Verbs

4.6 COBOL Tables

4.7 COBOL COPY Statement

4.8 COBOL REDEFINES

4.9 COBOL Intrinsic Functions

4.10 COBOL Control Break Processing

4.11 Example COBOL Data Validation Program

4.1 Introduction to COBOL

Shortly after Grace Hopper, the "Grandmother of Programming," designed her own English-like language, FLOW-MATIC for the computer on which she and her team were working, the Committee on Data Systems Languages (CODASYL) was formed in 1959 to study and develop a standard programming language that could be used across platforms at that time. Their efforts led to the development of COBOL

COBOL is an acronym for COmmon Business-Oriented Language. Like mainframes in general, the COBOL programming language has been mistakenly thought of as being outdated – NOT TRUE! In the 1990s alone, billions of dollars were spent on COBOL code preparing it for the turn of the century between 1999 and 2000, or "Y2K". As stated in Murach's *Introduction to COBOL Programming*, many more billions of dollars will be spent on the enhancement and maintenance of the same programs. In fact, in 2002, an object-oriented version of COBOL was introduced and can even be programmed for the .NET platform.

Although there are many aspects of COBOL that we could study, our focus will be on what could be called the basics of the language. One aspect of COBOL is often used to retrieve data from and update DB2 or other large database structures using SQL statements while running under the Customer Information Control System, or CICS, thus allowing fast and extremely dependable access to stored data in real time via internet-based or other types of GIU-based applications. It is easier to learn the basics of COBOL, though, by writing what are commonly referred to as batch programs that read data from files and formatting the data into output reports. It is precisely this type of programming to which you will be introduced in this chapter.

After learning languages like C++, Java or even Assembly, learning COBOL is relatively easy. Of course, there are some quirks in the language that may take some time to get used to. As stated above, the language was the first developed after Assembly. It was initially supposed to be a programming language that could be learned and used to develop data processing programs by the average office secretary of the day, hence the English-like sentences and the use of verbs and periods found in COBOL. But it was soon obvious that it required more knowledge about programming than an office secretary could ever be expected to have.

COBOL used to be a language that required all source code to be typed in capital letters but now allows a combination of upper and lower case letters. Because of this, it is considered case "insensitive", or it is not case sensitive. Although this is the case, all COBOL in this text will be presented in all capital letters as it is, in many ways, easier to scan with our eyes and read quickly. You are also required to write all of your COBOL in capital letters in this class.

4.2 COBOL Program Outline

This is just an overview of how a COBOL program is structured.

The Four Divisions of a COBOL Program

- `IDENTIFICATION DIVISION.`

  Must begin in column 8. This provides identifying information about the program. It must include:

```
PROGRAM-ID.
```

  and may also include some other information (all optional) that all begin in column 8:

```
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.
```

- `ENVIRONMENT DIVISION.`

  Must begin in column 8. What is included in this division depends on the specific computer system being used. Some of the details of the format will vary accordingly. For instance, if using VSAM, the `SELECT ASSIGN` statements become more complex.

  The `INPUT-OUTPUT SECTION` must begin in column 8 and provides information about the files in use in the program and is needed if the program uses one or more files, whether reading from or writing to.

  It contains:

  `FILE-CONTROL.`

  `SELECT ASSIGN` statements

  Example:

```
           SELECT INPUT-FILE ASSIGN TO INFILE.
```

  The `SELECT ASSIGN` statements begin in column 12 or after. It links the name of the file, `INPUT-FILE`, as manipulated by statements in the COBOL program with the DD name of the file in the JCL, `INFILE`.

- `DATA DIVISION.`

  Must being in column 8. This describes the data items the program will use, i.e., variables, column and page header definitions, etc. It can also contain the layouts of incoming and outgoing records, the current date and time fields used by the intrinsic date function and more. It contains several sections that must all begin in column 8. Each of the sections is technically optional. They are:

`FILE SECTION.`

The `FILE SECTION` describes the files in use, along with the definition, or layout, of what the data record looks like for those files. It is needed if the program reads from and/or writes to files of any type.

`WORKING-STORAGE SECTION.`

The `WORKING-STORAGE SECTION` defines and, in many cases, initializes the variables. It is necessary if there are variables necessary to be declared.

`LINKAGE SECTION.`

The `LINKAGE SECTION` describes parameters used when the program is called by another program (and may contain other items as well). It is also necessary if the program receives a `PARM` on the `EXEC` card at time of execution.

`LOCAL-STORAGE SECTION.`

The `LOCAL-STORAGE SECTION` is used in recursion and is not needed otherwise.

- `PROCEDURE DIVISION.`

  Must begin in column 8. This contains all of the executable code, normally organized in a main routine and assorted internal subroutines or "paragraphs" as they are called in COBOL.

Example

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    Program name goes here
AUTHOR.        Programmer's name goes here
DATE-WRITTEN.  The due date or date finished goes here
DATE-COMPILED. (Leave blank as it is supplied by the system.)

ENVIRONMENT DIVISION.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

    SELECT file-name ASSIGN TO ddname.

DATA DIVISION.

FILE SECTION.

FD  file-name
    RECORDING MODE IS F.
```

```
01  record-name.
    Definition of record-name goes here

WORKING-STORAGE SECTION.

    Definitions of various items go here

PROCEDURE DIVISION.

0000-MAIN.

    COBOL code goes here

    GOBACK.

0000-EXIT. EXIT.

0100-SUBROUTINE.

    More COBOL code goes here

0100-EXIT. EXIT.
```

In the `PROCEDURE DIVISION` above, execution will begin with the first line of code, and it ends when the `GOBACK` command is executed. *There should only be a single `GOBACK` in any single program.*

This example has no documentation other than parenthetical remarks. As with any program written, documentation is very important, especially in long and complicated programs.

COBOL Code Layout

COBOL code is written in 80-column lines. Leave columns 1-6 and 73-80 blank, though.

In COBOL, columns 8-11 are referred to as the A margin and columns 12-72 as the B margin.

Column 7 is used in two ways. If an asterisk (*) is placed in column 7, the remainder of the line is considered documentation and ignored by the compiler. Of course, do not go past column 72. If a hyphen (-) is placed in column 7, it indicates the continuation of a character literal from the line immediately above.

Once again, the names of the four divisions, the names of sections, the names of subroutines, 01-level items in the `DATA DIVISION`, etc., must all begin in column 8.

4.3 COBOL Numeric Formats

For application programmers, there are basically four ways to declare storage for numeric data in COBOL. The USAGE IS clause used to be necessary to denote how numbers are being stored in a single elementary item but USAGE IS can now be omitted in the declaration of storage. Indication of the format is still required unless the default of DISPLAY, or zoned decimal format, is desired.

The term DISPLAY here is a bit misleading, though. Remember that zoned decimal numbers can be signed. If the zone digit of the rightmost byte of the field, i.e., the next to last hexadecimal digit, is F, A, C or E, the number is positive. If it is B or D, the number is negative. For the number to be purely DISPLAY, it would have to be a positive number with the rightmost byte's zone digit as F so that it is actually EBCDIC.

Example

Here are four different ways in which the number 23156 can be stored in zoned decimal format, two hexadecimal digits per byte:

```
F2F3F1F5F6
F2F3F1F5A6
F2F3F1F5C6
F2F3F1F5E6
```

The only one of the four that is truly DISPLAY format is the first. If you displayed, for example, the third of these representations above, you would see the following:

2315F because hex C6 is EBCDIC for the capital letter F.

The Four COBOL Numeric Formats

DISPLAY	a zoned decimal number 1 character/byte default format
PACKED-DECIMAL or COMP-3	a packed decimal number 2 digits/byte last byte contains the sign digit
BINARY or COMP or COMP-4	a binary number S9 to S9(4) – stored as 2 bytes S9(5) to S9(9) – stored as 4 bytes S9(10) to S9(18) – stored as 8 bytes NOT aligned on a specific boundary
BINARY SYNC or COMP SYNC or COMP-4 SYNC	a binary number on a specific boundary S9 to S9(4) – stored as 2 bytes on a halfword boundary S9(5) to S9(9) – stored as 4 bytes on a fullword boundary S9(10) to S9(18) – stored as 8 bytes on a doubleword boundary

Note that COMP is an abbreviation for COMPUTATIONAL.

For the sake of being up-to-date, please use PACKED-DECIMAL, BINARY and BINARY SYNC in your COBOL programs.

The `USAGE IS` part of the phrase is optional and ***should be left out***. It is better to simply do the following:

```
       01  FIELD-B     PIC S9(5)  COMP-3.
```

or

```
       01  FIELD-B     PIC S9(5)  PACKED-DECIMAL.
```

Of course, the items could also have `VALUE` clauses:

```
       01  FIELD-B     PIC S9(5)  COMP-3  VALUE 327.
```

or

```
       01  FIELD-B     PIC S9(5)  PACKED-DECIMAL  VALUE 8433.
```

As a review, these are stored in different ways. Remember from above that the default is DISPLAY format, which is zoned decimal numbers, one per byte (or column).

Although zoned decimal number can actually be EBCDIC for digits 0 through 9 (F0 through F9), as stated above, do NOT assume they are. For example, either of the following five-byte fields holds the zoned decimal representation of -354:

`F0F0F3F5B4` or `F0F0F3F5D4`

Although there is no character match for EBCDIC `B4`, there is for EBCDIC `D4`; it is the capital letter M.

`COMP-3`, or `PACKED-DECIMAL`, means that the value will be stored as a packed decimal number, i.e., one byte for each two digits, including a sign digit at the end.

`COMP`, `COMP-4`, or `BINARY`, means that the value will be stored in binary form of one of three lengths:

`S9` to `S9(4)` is stored as 2 bytes, a half word
`S9(5)` to `S9(9)` is stored as 4 bytes, a full word
`S9(10)` to `S9(18)` is stored as 8 bytes, a double word

Hint: Just remember "1, 5 and 10" and "2, 4 and 8" and it is easy to remember how long the field actually is in bytes.

These refer to 2, 4 or 8 bytes in binary format, not necessarily to anything declared as a half word, full word or double full word; i.e., not necessarily aligned on a boundary.

`COMP SYNC`, `COMP-4 SYNC`, or `BINARY SYNC`, means the same as `COMP` but the storage is aligned on a half word boundary (for `S9` to `S9(4)`) or on a full word boundary (otherwise).

Because of the SYNC alignment, there might be slack bytes embedded in `WORKING-STORAGE`. This can affect calculating the size of a table entry, and being aware of it is important when doing group moves, computing the size of a record, or passing parameters to Assembler programs.

Once again, note that anything declared at a 01-level in the COBOL Data Division is stored on a doubleword boundary.

One of the very best characteristics of COBOL is that it performs the conversions between numeric formats in elementary moves and arithmetic operations. It is important to remember that it will NOT perform the conversions between numeric formats if moved with `MOVE CORRESPONDING`, though! More on this later.

More Examples of the Four Formats

Display Examples

```
       01  FLD-A   PIC  S9(5).
```

or

```
       01  FLD-A   PIC  S9(5)   DISPLAY.
```

Packed Decimal Examples

```
       01  FLD-B   PIC  S9(5)   PACKED-DECIMAL.
```
← Preferred notation.

or

```
       01  FLD-B   PIC  S9(5)   COMP-3.
```

or

```
       01  FLD-B   PIC  S9(5)   COMPUTATIONAL-3.
```

Binary Examples

```
       01  FLD-C   PIC  S9(5)   BINARY.
```
← Preferred notation.

or

```
       01  FLD-C   PIC  S9(5)   COMP.
```

or

```
       01  FLD-C   PIC  S9(5)   COMPUTATIONAL.
```

or

```
       01  FLD-C   PIC  S9(5)   COMP-4.
```

or

```
       01  FLD-C   PIC  S9(5)   COMPUTATIONAL-4.
```

Binary On Boundary Examples

```
       01  FLD-D   PIC  S9(5)   BINARY SYNC.
```
← Preferred notation.

or

```
       01  FLD-D   PIC  S9(5)   COMP SYNC.
```

or

```
       01  FLD-D   PIC  S9(5)   COMP-4 SYNC.
```

Usage Guidelines

`DISPLAY` Numerical items that will not be used for arithmetic and will be printed.

```
        01  FLD-A   PIC  S9(5)   DISPLAY  VALUE 23.
```

or

```
       01  FLD-A   PIC  S9(5)   VALUE 23.
```

stored as 5 bytes: `F0 F0 F0 F2 F3`

Note: As stated above, the sign does NOT take an additional byte of storage but rather is noted by the zone hex digit in the final, or rightmost, byte of the representation. F, A, C, or E are positive representatios and B and D are negative.

`PACKED-DECIMAL/COMP-3` Numerical items that will be used in arithmetic and will eventually be printed.

```
    01  FLD-B   PIC  S9(5)   PACKED-DECIMAL  VALUE 23.
```

or

```
    01  FLD-B   PIC  S9(5)   COMP-3  VALUE 23.
```

stored as 3 bytes: `00 02 3F`

`BINARY/COMP/COMP-4` Numerical items that will be used ONLY for arithmetic and will NOT be printed.

```
    01  FLD-C   PIC  S9(5)   BINARY.
```

or

```
    01  FLD-C   PIC  S9(5)   COMP  VALUE 23.
```

or

```
01  FLD-C   PIC  S9(5)   COMP-4  VALUE 23.
```

stored as 4 bytes: `00 00 00 17`

Suggested Usage Rules

Use `DISPLAY` for numeric items that will not be used in arithmetic and will be printed. (No conversion will be needed to print them.)

Use `PACKED-DECIMAL` or `COMP-3`, for numeric items that will be used in arithmetic and will eventually be printed. (One conversion will be needed to print them.)

Use `BINARY` or `COMP`, for numeric items that will be used in arithmetic and will not be printed. (Two conversions will be needed to print them.)

Example of How the Formats are Stored:

Positive Integers

Suppose we have the items `FIELD-A`, `FIELD-B`, and `FIELD-C` as defined above, but each with the value 23:

```
01  FIELD-A     PIC S9(5)                   VALUE 23.
01  FIELD-B     PIC S9(5)  PACKED-DECIMAL   VALUE 23.
01  FIELD-C     PIC S9(5)  BINARY           VALUE 23.
```

Then:

`FIELD-A` might be stored as 5 bytes: `F0 F0 F0 F2 C3`
or `F0 F0 F0 F2 F3`
or `F0 F0 F0 F2 A3`
or `F0 F0 F0 F2 E3`

`FIELD-B` is stored as 3 bytes: `00 02 3C`
or `00 02 3F`
or `00 02 3A`
or `00 02 3E`

`FIELD-C` is stored as 4 bytes: `00 00 00 17`

For `FIELD-C`, remember that the leftmost bit of the 32 is the sign bit. A 0 indicates a positive binary representation and a 1 indicates a negative.

Negative Integers

Suppose we move the value -23 to each of the same three fields defined above.

Then:

FIELD-A is stored as 5 bytes: F0 F0 F0 F2 D3
or F0 F0 F0 F2 B3

FIELD-B is stored as 3 bytes: 00 02 3D
or 00 02 3B

FIELD-C is stored as 4 bytes: FF FF FF E9

FIELD-C as a 4-byte 32-bit representation of -23:

```
11111111 11111111 11111111 11101001
^
```

the sign bit

For FIELD-C, remember that the negative bit representation of an integer, as shown above, is the two's complement of the bit representation of the integer's absolute value.

Note that, if the S denoting signed storage is omitted from a numeric picture clause definition, the storage will not be able to store anything but the absolute value of any negative number.

Also remember that the sign of any number stored in signed storage will NOT take any storage itself but is inherent in how the number is stored itself.

4.4 COBOL Numeric-Edited Output ← under construction!

COBOL facilitates the creation of formatted numeric-edited output.

The Floating Dollar Sign

Placiong a floating dollar sign just to the left of the first non-zero digit when reading the number from left to right and with commas appropriately spaced is simple in COBOL.

For example, an accumulator field defined as:

```
05  ACCT-TOTAL          PIC S9(12)V99  PACKED-DECIMAL  VALUE 0.
```

can be easily moved to a field in which it will be formatted with a floating dollar sign and commas. If the output field is defined as:

```
05  OUT-ACCT-TOTAL      PIC $$$$,$$$,$$$,$$9.99.
```

If a dollar amount has been accumulated in `ACCT-TOTAL`, the statement:

```
MOVE ACCT-TOTAL TO OUT-ACCT-TOTAL.
```

will result in the value being fully formatted with a floating dollar sign placed just to the left of the first non-zero digit (from left to right) and with commas appropriately spaced.

For example, if the field `ACCT-TOTAL` holds the value 46533.87, the displayed value would be:

```
bbbbbbbbb$46,533.87
```

where the lower-case letter b is used to denote a space.

If the possibility exists that a negative dollar value will be moved to the numeric-edited output field, either of the following definitions should be used:

```
05  OUT-ACCT-TOTAL      PIC $$$$,$$$,$$$,$$9.99DB.
```

or

```
05  OUT-ACCT-TOTAL      PIC $$$$,$$$,$$$,$$9.99CR.
```

If a negative value is moved to either of these two example fields, the `DB` (debit) or `CR` (credit) will be displayed, respectively. If the value is positive, the `DB` or `CR` will be replaced by two spaces. This may seem strange because a credit indicates positive to most uf us but, in accountancy, both a debit and a credit are considered negative values. Note that it is best to only use the DB or CR when using the floating dollar point or, more specifically, when displaying dollar amounts.

4.5 COBOL Verbs

Commands in COBOL are known as COBOL verbs. Because COBOL is so like structured English and was first implemented for non-programmers to use for simple programming, the term *verb* is appropriate.

The most commonly used COBOL verbs and some of their variations are:

`OPEN`

- Used to open files, i.e., data sets, for processing in the COBOL program.
- In most cases, either `OPEN INPUT` or `OPEN OUTPUT` is used because sequential files and PDS or PDSE members can only be accessed one direction at a time.
- `OPEN` ***can*** be used without `INPUT` or `OUTPUT` when accessing some files, though, i.e., database files that can be open for reading from or writing to simultaneously.

`OPEN INPUT`

- Used to open files for reading records from.
- Example:
```
OPEN INPUT TRANSACTION-FILE.
```
- Note that a single `OPEN` can be used to open one or more files at once:
- Example:
```
OPEN INPUT TRANSACTION-FILE
     OUTPUT REPORT-FILE.
```

  Note that the indentation lining up `INPUT` and `OUTPUT` is for readability. Also note the placing of the period at the end of the command.

`OPEN OUTPUT`

- Used to open files for writing records to.
- Example:
```
OPEN OUTPUT REPORT-FILE.
```
- As stated and illustrated above, a single `OPEN` can be used to open one or more files – and in different directions – at once.

`CLOSE`

- Used to close files when processing of the data set is finished.
- All files that are open should be closed before ending the COBOL program.
- Example:
```
CLOSE TRANSACTION-FILE.
```
- Note that a single `CLOSE` can be used to close one or more files at once:

- Example:
```
CLOSE TRANSACTION-FILE
      REPORT-FILE.
```

  Note that the indentation lining up the names of the files is for readability. Also note the placing of the period at the end of the command.

READ

- Used to read a single record from a file.
- The file must be open for input prior to the `READ`.
- Example:
```
READ TRANSACTION-FILE.
```

  or
```
READ TRANSACTION-FILE
END-READ.
```

- If a record is successfully read, the definition of the record under the `01`-level in the file's `FD` can be used to access the various fields in the record itself.

READ INTO

- Used to read a single input record from a file and copy it into, in most cases, a `01`-level item defined in the `WORKING-STORAGE SECTION`.
- Example:
```
READ TRANSACTION-FILE INTO TRANS-RECORD.
```

  or
```
READ TRANSACTION-FILE INTO TRANS-RECORD
END-READ.
```

READ AT END

- Used when reading one input record at a time from a file with multiple records in a read until end-of-file loop.
- Example:
```
READ TRANSACTION-FILE
  AT END MOVE 'Y' TO EOF-FLAG.
```

  or
```
READ TRANSACTION-FILE
  AT END MOVE 'Y' TO EOF-FLAG
END-READ.
```

- Note that `EOF-FLAG` is declared in the `WORKING-STORAGE SECTION` as:

```
01  EOF-FLAG           PIC X    VALUE 'N'.
```

  or it can be declared as a group item at the `05`-level or lower.

READ INTO AT END

- Like READ AT END, it is used when reading one input record at a time from a file with multiple records in a "read loop."

- And like READ INTO, the READ INTO AT END places a copy of the input record into, in most cases, a 01-level item defined in the WORKING-STORAGE SECTION.

- Example:
```
READ TRANSACTION-FILE INTO IN-ADD-RECORD
  AT END MOVE 'Y' TO EOF-FLAG.
```
  or
```
READ TRANSACTION-FILE INTO IN-ADD-RECORD
  AT END MOVE 'Y' TO EOF-FLAG
END-READ.
```

- In this example, IN-ADD-RECORD is a 01-level item in the WORKING-STORAGE SECTION.

- Again, the EOF-FLAG is declared in WORKING-STORAGE SECTION as:

```
01  EOF-FLAG           PIC X    VALUE 'N'.
```

  or it can be declared as a group item at the 05-level or lower.

WRITE

WRITE FROM
WRITE AFTER
WRITE FROM AFTER

PERFORM
PERFORM UNTIL
PERFORM VARYING UNTIL

MOVE

- Used to move – actually copy – data from one place in storage to another.

- Moving a single item to another is the most common use and is known as an "elementary move."

- If a numeric conversion, i.e., DISPLAY to PACKED-DECIMAL, is required, the move statement forces the conversion to occur.

- If moving a numeric item to a numeric-edited field, the move statement forces the conversion and editing to occur.

- Example:

```
MOVE IN-ACCT-BAL TO ACCT-BAL.
```

where IN-ACCT-BAL is defined as an elementary item in an input record layout:

```
01  IN-ACCT-RECORD.
    05  IN-ACCT-NBR         PIC S9(16).
    05  IN-ACCT-BAL         PIC S9(13)V99.
    etc.
```

and ACCT-BAL is defined as an elementary item variable used for arithmetic in the WORKING-STORAGE SECTION:

```
01  VARIABLES.
      05  ACCT-BAL          PIC S9(13)V99 PACKED-DECIMAL VALUE 0.
```

In this example, IN-ACCT-BAL, an elementary item in DISPLAY format, is copied to an elementary item named ACCT-BAL in PACKED-DECIMAL format and the conversion from DISPLAY to PACKED-DECIMAL is automatic.

- Example:

```
MOVE ACCT-BAL TO OUT-ACCT-BAL.
```

where ACCT-BAL is defined in the WORKING-STORAGE SECTION as:

```
01  VARIABLES.
    05  ACCT-BAL            PIC S9(13)V99  PACKED-DECIMAL  VALUE 0.
```

and OUT-ACCT-BAL is defined in the WORKING-STORAGE SECTION as an elementary item of a 132-byte output record layout named OUT-ACCT-DETAIL:

```
01  OUT-ACCT-DETAIL.
    .
    .
    05  OUT-ACCT-BAL        PIC $$,$$$,$$$,$$$,$$9.99.
```

MOVE CORRESPONDING

- In COBOL, duplicate names can be used as long as they are

```
ADD
ADD GIVING
ADD GIVING ROUNDED
SUBTRACT
SUBTRACT GIVING
SUBTRACT GIVING ROUNDED
MULTIPLY
MULTIPLY GIVING
MULTIPLY GIVING ROUNDED
DIVIDE
DIVIDE GIVING
DIVIDE GIVING ROUNDED
```

```
COMPUTE
COMPUTE ROUNDED

SEARCH
SEARCH ALL

INSPECT REPLACING

EVALUATE
```

- Unless writing records to a file, never use `WRITE` without the `AFTER` for line spacing.
- Never `WRITE` blank lines.
- Unless you have adequate reason, do NOT use the `READ AT END` with the `NOT AT END` structure.
- If possible, do not use the arithmetic verbs above except if adding to a counter of some sort. For example: `ADD 1 TO LINE-CTR.` For all others, use `COMPUTE` and `COMPUTE ROUNDED` below.

4.6 COBOL Tables

COBOL tables are defined in the DATA DIVISION using the OCCURS n [TIMES] clause. If the table level is to be indexed, both the OCCURS n [TIMES] and the INDEXED BY clauses must be used together in the table's definition.

If using a subscript to reference a table entry, simply define it in WORKING-STORAGE as an integer large enough to hold the maximum number of entries. Always define a subscript as BINARY SYNC, COMP SYNC or COMP-4 SYNC. For example, if a table has an OCCURS 500, the subscript could be defined as:

```
01  SUBSCRIPTS.
    05  TBL-SUB          PIC S9(3)   BINARY SYNC.
```

or

```
01  TBL-SUB              PIC S9(3)   BINARY SYNC.
```

Unlike subscripts, the INDEXED BY clause simply gives a name to an index and the named index requires no further definition. In other words, and index will have **NO** PIC clause anywhere in the program.

If using an index, it cannot be manipulated by the standard arithmetic verbs (ADD, SUBTRACT,...) or the MOVE verb. Instead, the SET verb must be used:

FORMAT 1: SET var_name_1/index_1 TO var_name_2/index_2/integer

FORMAT 2: SET index_1 index_2 ... UP/DOWN BY var_name/integer

Like subscripts, index values can also be altered by PERFORM VARYING, SEARCH, or SEARCH ALL verbs.

Indexes are stored as binary values that represent the displacement from the beginning of the table to the specific entry.

Unlike third, fourth and fifth-generation languages and arrays, the valid range for both subscripts and indexes is 1 to n with n being the maximum number of entries. Of course, because indexes are stored as byte displacements, an index value of 1 corresponds to a byte displacement of 0.

A multi-dimensional table is created by nesting OCCURS clauses.

```
01  ANNUAL-SALES-DATA.
    05  YEARLY-SALES           OCCURS 5
                                INDEXED BY YR-NDX.
        10  MONTHLY-SALES      OCCURS 12
                                INDEXED BY M-NDX.
            15  SALE-AMOUNT    PIC  9(7)V99.
```

To access an entry in a multi-dimensional table, the order of the indexes will correspond to the nesting order of the OCCURS clauses.

SALE-AMOUNT(1, 4) references year 1 and month 4 (April).

`SALE-AMOUNT(4, 1)` references year 4 and month 1 (January).

To access an entry in a multi-dimensional table, every field needs as many indexes as there are `OCCURS` clauses that are a part of it or above it. In other words:

`SALE-AMOUNT(4)` would be invalid and will not compile.

Example of Multi-Dimensional Table Processing

The following is defined `WORKING-STORAGE`:

```
01  ANNUAL-SALES-DATA.
    05  YEARLY-SALES           OCCURS 5
                                INDEXED BY YR-NDX.
        10  MONTHLY-SALES      OCCURS 12
                                INDEXED BY M-NDX.
            15  SALE-AMOUNT    PIC  9(7)V99.
```

The following are statements in the `PROCEDURE-DIVISION`:

```
0000-MAIN.
    .
    .     other statements can precede the following
    .
    PERFORM VARYING YR-NDX FROM 1 BY 1 UNTIL YR-NDX > 5
      PERFORM VARYING M-NDX FROM 1 BY 1 UNTIL M-NDX > 12
        ADD SALE-AMOUNT(YR-NDX, M-NDX) TO TOTAL-AMOUNT
      END-PERFORM
    END-PERFORM.
    .
    .
    GOBACK.

0000-EXIT. EXIT.
```

or

```
0000-MAIN.
    .
    .     other statements can precede the following
    .
    PERFORM 1000-ADD VARYING YR-NDX FROM 1 BY 1 UNTIL YR-NDX > 5
      AFTER M-NDX FROM 1 BY 1 UNTIL M-NDX > 12.
    .
    .
    GOBACK.

0000-EXIT. EXIT.
```

```
1000-ADD.

    ADD SALE-AMOUNT(YR-NDX, M-NDX) TO TOTAL-AMOUNT.

1000-EXIT. EXIT.
```

Both of the above examples would accomplish the same thing.

COBOL Subscripts vs. Indexes

Differences Between Subscripts and Indexes:

Subscripts	***Indexes***
1. Needs to be declared with `PIC` clause.	1. Does NOT need to be declared with a `PIC` clause
2. Holds a binary value of 1 to n.	2. Holds a binary value representing a displacement in bytes from the beginning of the table.
3. Can be displayed using `DISPLAY`.	3. Cannot be displayed using `DISPLAY`.
4. Can be altered using the arithmetic verbs.	4. Can only be altered using the `SET` verb.
5. Can be altered using the `MOVE` verb.	5. Cannot be altered using the `MOVE` verb.
6. Can refer to any entry at any level of any tree.	6. Can only refer to an entry at a specific level of a specific tree.
7. Cannot be used with the `SEARCH` or `SEARCH ALL` verbs.	7. Required to be used with the `SEARCH` or `SEARCH ALL` verbs.

Similarities Between Subscripts and Indexes:

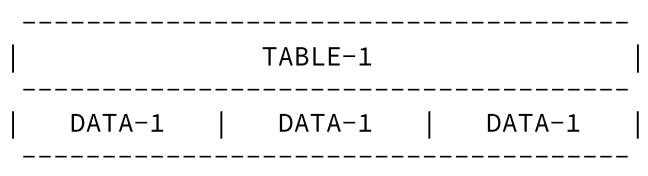
- Both reference a single entry at a single level of a COBOL table.
- Can be altered, or used, with the `PERFORM VARYING` with no special considerations.

Note: Subscripts should always be defined as `BINARY SYNC` or `COMP SYNC` and, for good form, a subscript name should end with the suffix `-SUB` and an index with the suffix `-NDX`.

Examples of COBOL Table Storage

One- or Single-Dimensional Table:

```
01  TABLE-1.
    05  DATA-1      OCCURS 3    PIC X(6).
```



Another One- or Single-Dimensional Table:

```
01  TABLE-2.
    05  DATA-2        OCCURS 3.
```

```
           10  FLD-A     PIC X(4).
           10  FLD-B     PIC X(4).
```

```
 ------------------------------------------------
|                     TABLE-2                    |
 ------------------------------------------------
|     DATA-2    |     DATA-2    |     DATA-2    |
 ------------------------------------------------
| FLD-A | FLD-B | FLD-A | FLD-B | FLD-A | FLD-B |
 ------------------------------------------------
```

Two-Dimensional Table:

```
       01  TABLE-3.
           05  FLD-A         OCCURS 3.
               10  FLD-B     OCCURS 2
                             PIC X(4).
```

```
 ------------------------------------------------
|                   TABLE-3                      |
 ------------------------------------------------
|     FLD-A     |     FLD-A     |     FLD-A     |
 ------------------------------------------------
| FLD-B | FLD-B | FLD-B | FLD-B | FLD-B | FLD-B |
 ------------------------------------------------
```

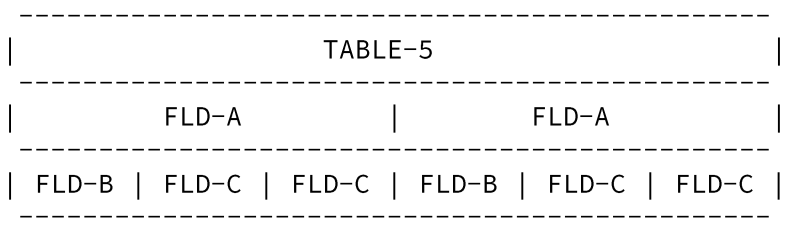
Two One- or Single-Dimensional Tables Under the Same 01-Level:

```
       01  TABLE-4.
           05  FLD-A         OCCURS 3
                             PIC X(3).
           05  FLD-B         OCCURS 2
                             PIC X(4).
```

```
 ------------------------------------------------
|                   TABLE-4                      |
 ------------------------------------------------
| FLD-A | FLD-A | FLD-A |   FLD-B   |   FLD-B   |
 ------------------------------------------------
```

Two-Dimensional Table:

```
       01  TABLE-5.
           05  FLD-A         OCCURS 2.
               10  FLD-B     PIC X(4).
               10  FLD-C     OCCURS 2
                             PIC X(4).
```



Note that any of the above-defined table examples can be indexed along with the `OCCURS` clause.

Also note that indexed table entries can be manipulated using a subscript as long as the subscript is large enough to hold the maximum number of entries for that table level. Indexes cannot be used to manipulate other levels of other tables.

4.7 COBOL SEARCH and SEARCH ALL Verbs

COBOL provides a means by which a table level can be searched for a matching value in a field without necessarily implementing a `PERFORM VARYING`. `SEARCH` is used to implement a sequential search through a table or a portion of a table and `SEARCH ALL` is used to implement a binary search.

Requirements of `SEARCH`

- Table levels on which a `SEARCH` will be implemented must be indexed.
- `SEARCH` can be implemented to look for a match in any field on the indexed level as long as the field is not itself part of a lower level table.
- The index value must be set before the `SEARCH` is implemented.
- The table level need not be in any sort order based on the field being searched.

Requirements of `SEARCH ALL`

- As with `SEARCH`, table levels on which a `SEARCH ALL` will be implemented must be indexed.
- The table level being searched with `SEARCH ALL` must be declared using the `ASCENDING KEY` or `DESCENDING KEY` field.
- For it to work, the table level must actually be in order based on the `ASCENDING` or `DESCENDING KEY` field.
- Unlike with `SEARCH`, the index value need not be set before the `SEARCH ALL` is implemented. In fact, if it is set, `SEARCH ALL` immediately alters in to begin the search in a binary fashion.

Examples

Given the following table definition:

```
01  FUND-TBL.
    05  FUND-TBL-ENTRY          OCCURS 300
                                ASCENDING KEY FUND-NBR
                                INDEXED BY FUND-NDX.
        10  FUND-NBR            PIC 9(2).
        10  FUND-SHR-PRC        PIC S9(3)V99 DECIMAL.
```

4.8 COBOL COPY Statement

The COBOL COPY statement facilitates the copying of previously written COBOL source code into a program. Although it is often referred to as the "COPY verb," the COPY statement is NOT a verb like MOVE, ADD, COMPUTE, READ, etc. The copying of the source code actually happens prior to COBOL compilation and is therefore NOT executable.

Although it is possible to copy in an entire COBOL program using the COPY statement, it is often used to copy in what is commonly referred to as a "copybook." This is a section of storage like that found in the DATA DIVISION with levels and PIC clauses. But, as stated above, the code copied in can also be COBOL statements, COBOL paragraphs, or anything that can be typed into a COBOL program.

COBOL copybooks are useful in a number of ways but several of their primary uses are 1) to save the programmer time in typing in lots of large, multi-leveled and/or complex DATA DIVISION statements and 2) to facilitate the use of conventional or standard names for fields between programs in the same application or between applications. For example, a mutual fund transaction processing system's conventional – or standardized – name for the field that stores the shareholder's legal name might be SHR-HOLDER-NME and have size of PIC X(30). If copybooks are created – and used – for every DATA DIVISION structure in every COBOL program in the system or application that uses the shareholder's legal name, the same elementary item name and size will be used. This can potentially save a lot in maintenance and any future enhancements. Using copybooks, imagine how much easier it would be to scan an application's COBOL program source code for the standardized name and, if necessary, expand the field to PIC X(35).

When using the COPY statement, the source of the line or lines of code that are copied into a program is a single member of what is usually referred to as a COPYLIB, or copy library, PDS or PDSE. Copy libraries meant to be used in COBOL programs have 80-byte-long records that are inserted in whole in the program immediately following the COPY statement itself.

Format:

```
COPY member-name
           [SUPPRESS]
           [REPLACING string1 BY string2
                      string3 BY string4].
```

member-name PDS or PDSE member name of the code to copy into the program.

SUPPRESS the copied text will not be printed in the source code listing.

REPLACING replaces every occurrence of string1 by string2.

Note that the text that is copied in will have a 'C' in column 1 of the source listing unless suppressed.

Example:

In the COBOL Compiler step, add a DD card named SYSLIB referencing the PDS or PDSE where the copybooks are stored as members.

Example:

```
//SYSLIB   DD DSN=KC0nnnn.CSCI465.COPYLIB,DISP=SHR
```

Note that copy library PDSs or PDSEs can be concatenated on the SYSLIB card if necessary.

In the DATA DIVISION, or anywhere else in your program you want to copy the code from the PDS or PDSE member, use the COPY statement such as:

```
    COPY SALESREC.
```

Each 80-byte line from the copy library member named SALESREC will be inserted in the source code following the COPY statement itself. Once again, this happens prior to compilation.

In Assembler, the COPY statement works almost exactly the same way as in COBOL although the SUPPRESS and REPLACING options cannot be used and there is no need for the LIB parameter on the EXEC card for the Assembler. The COPY library is included in SYSLIB as before because this is very similar to using a macro. Note that macro libraries are also in SYSLIB.

In COBOL or in Assembler, the code to be copied may contain other COPY statements, but no circular COPY references are allowed, and this practice is strongly discouraged in any case.

4.9 COBOL REDEFINES

As the name suggests, the COBOL `REDEFINES` allows for the redefinition of any storage defined in the `DATA DIVISION`. One might consider it a new way to look at the same bytes.

As an example of the use of the `REDEFINES`, the following COBOL data structure involves redefining sections of an input record that can come in several different formats. This is a typical situation and a good example as a single transaction input record can represent one of several types of transactions.

The COBOL program reads a record from the input file and then, based on the transaction code, determines the type of transaction to process and thus which data items to use based on that transaction type. Note that the `PIC X(18)` field originally defined as `IN-DATA` is being redefined in three different ways. It is the same 18 bytes of storage but the `REDEFINES` allows the programmer to consider those bytes in several different formats based on the `IN-TRAN-CODE`.

```
01  IN-TRAN-RECORD.
    05  IN-TRAN-CODE                PIC X.
        88  ADD-TRANSACTION         VALUE 'A'.
        88  UPDATE-TRANSACTION      VALUE 'U'.
        88  DELETE-TRANSACTION      VALUE 'D'.
    05  IN-TRAN-FIELD               PIC 9(6).
    05  IN-DATA.                    PIC X(18).
    05  IN-ADD-TRAN REDEFINES IN-DATA.
        10  IN-ADD-FIELD1           PIC X.
        10  IN-ADD-FIELD2           PIC 9.
        10  IN-ADD-FIELD3           PIC X(10).
        10  IN-ADD-FIELD4           PIC 9(6).
    05  IN-ALTER-TRAN REDEFINES IN-DATA.
        10  IN-ALTER-CODE           PIC 9.
        10  IN-ALTER-DATA           PIC X(17).
        10  IN-ALTER-1 REDEFINES IN-ALTER-DATA.
            15  IN-ALTER-FIELD1     PIC 9(3).
            15  FILLER              PIC X(14).
        10  IN-ALTER-2 REDEFINES IN-ALTER-DATA.
            15  IN-ALTER-FIELD2     PIC 9(8).
            15  FILLER              PIC X(9).
        10  IN-ALTER-3 REDEFINES IN-ALTER-DATA.
            15  IN-ALTER-FIELD3     PIC X(10).
            15  FILLER              PIC X(7).
    05  DELETE-TRAN REDEFINES IN-DATA.
        10  FILLER                  PIC X(18).
```

Rules for the `REDEFINES`

A data item can only redefine a preceding item at the same level.

The number of bytes in the redefined area must match the number of bytes of the area being redefined.

Example

The next example involves defining a table already loaded with information. For good measure, it is redefined twice, once with subscripts and once with indexes.

```
01  SEASON-NAME-DATA.
    05  FILLER                 PIC X(6) VALUE 'SPRING'.
    05  FILLER                 PIC X(6) VALUE 'SUMMER'.
    05  FILLER                 PIC X(6) VALUE 'FALL'.
    05  FILLER                 PIC X(6) VALUE 'WINTER'.
01  SEASON-NAME-TBL            REDEFINES SEASON-NAME-DATA.
    05  SEASON-NAME            OCCURS 4
                               PIC X(6).
01  SEASON-NAME-TBL-NDXD       REDEFINES SEASON-NAME-DATA.
    05  SEASON-NAME-NDXD       OCCURS 4
                               INDEXED BY S-N-NDX
                               PIC X(6).
```

`VALUE` clauses can only be placed on the original data item definitions.

4.10 COBOL Intrinsic Functions

An intrinsic function performs a mathematical, character, or logical operation and allows the programmer to reference a data item that is derived from other data items during execution of the program.

Functions are elementary data items. They are classified as one of three types -- alphanumeric, numeric, or integer -- depending on the type of value they return. Functions may not be receiving operands.

A function is referenced by specifying its name, along with any required arguments, in a `PROCEDURE DIVISION` statement.

A few of the intrinsic functions are briefly described below. Date functions are described in the next section.

FUNCTION CURRENT-DATE

Probably the most useful intrinsic function is `CURRENT-DATE` which is a replacement for `ACCEPT DATE` and `ACCEPT TIME`. `CURRENT-DATE` is Y2K-compliant, having a 4-digit year. This function returns data to fill a user-defined 20-character alphanumeric field which is laid out as follows:

```
01  CURRENT-DATE-FIELDS.
    05  CURRENT-DATE.
        10  CURRENT-YEAR            PIC  9(4).
        10  CURRENT-MONTH           PIC  9(2).
        10  CURRENT-DAY             PIC  9(2).
    05  CURRENT-TIME.
        10  CURRENT-HOUR            PIC  9(2).
        10  CURRENT-MINUTE          PIC  9(2).
        10  CURRENT-SECOND          PIC  9(2).
        10  CURRENT-MS              PIC  9(2).
    05  DIFF-FROM-GMT               PIC S9(4).
```

Type or copy the above into your `WORKING-STORAGE SECTION`. You may change the names of the fields to suit your needs but the data types and field lengths must remain as described above. Also, it is useful to put this at the very end of your `WORKING-STORAGE SECTION` as you probably will not refer to the fields very often.

Notice that the time returned is given down to the millisecond and you are also provided the difference between your time and Greenwich Mean Time in the field named `DIFF-FROM-GMT` above. A negative value means that you are ahead of GMT and positive means you are behind.

The function is used in a `MOVE`:

```
MOVE FUNCTION CURRENT-DATE TO CURRENT-DATE-FIELDS.
```

You may also use reference modification to only grab the part of the date you want:

* Get the current date in yyyymmdd format

```
MOVE FUNCTION CURRENT-DATE (1:8) TO WS-TODAY.
```

* Get the current time in hhmmss format

```
MOVE FUNCTION CURRENT-DATE (9:6) TO WS-TIME.
```

Note that, in the above two move statements, the destination fields `WS-TODAY` and `WS-TIME` are user-defined fields.

If you can, call the `CURRENT-DATE` function once at the very beginning of your program. Whenever possible, avoid calling the function from within a loop.

FUNCTION LENGTH (argument-1)

The length function returns an integer equal to the length of the argument (in bytes).

The argument may be a nonnumeric literal or a data item.

Example: `COMPUTE LENGTH-OF-VARIABLE = FUNCTION LENGTH(VARIABLE-1).`

The data item `LENGTH-OF-VARIABLE` will contain the number of bytes in the data item `VARIABLE-1`.

FUNCTION MAX (argument-1 argument-2 ...)

The `MAX` function returns the content of whichever argument contains the greatest value.

The function type depends on the argument type.

Arguments may be numeric, alphanumeric, or alphabetic, but all arguments must be of the same type.

Example:
```
COMPUTE HIGH-NUMBER =
                          FUNCTION MAX (VARIABLE-1 VARIABLE-2 VARIABLE-3).
```

The data item `HIGH-NUMBER` will contain the same value as whichever variable has the highest value.

FUNCTION MIN(argument-1 argument-2 ...)

The `MIN` function returns the content of whichever argument has the smallest value.

The function type depends on the argument type.

Arguments may be numeric, alphanumeric, or alphabetic, but all arguments must be of the same type.

Example:
```
COMPUTE LOW-NUMBER =
                  FUNCTION MIN(VARIABLE-1 VARIABLE-2 VARIABLE-3).
```

The data item `LOW-NUMBER` will contain the same value as whichever variable has the lowest value.

FUNCTION MEAN(argument-1 argument-2 ...)

The `MEAN` function returns a numeric value that approximates the arithmetic average of the arguments.

The arguments must be numeric.

Example:
```
COMPUTE AVERAGE-VALUE =
                              FUNCTION MEAN(VAR-1 VAR-2 VAR-3 VAR-4).
```

The data item AVERAGE-VALUE will contain the average of the four variables, calculated by adding up the values in the arguments and dividing by the number of arguments.

The ALL subscript can be used to obtain the average of items contained in a table.

Example:
```
COMPUTE AVERAGE-SALARY = FUNCTION MEAN(EMP-SALARY(ALL)).
```

The values in all of the instances of EMP-SALARY in the table will be added together, and the total will be divided by the number of instances. The result will be placed in AVERAGE-SALARY.

FUNCTION MEDIAN(argument-1 argument-2 ...)

The MEDIAN function returns the middle value in a list formed by arranging the arguments in sorted order. If the number of occurrences is even, the returned value is the arithmetic mean of the values referenced by the two middle occurrences.

The value returned is a numeric value, and all of the arguments must be numeric.

Example:
```
COMPUTE MIDDLE-NUMBER =
                          FUNCTION MEDIAN(VAR-1 VAR-2 VAR-3 VAR-4 VAR-5).
```

MIDDLE-NUMBER will contain the median of the five variables used as arguments.

The ALL subscript could be used to obtain the median of a series of numbers stored in a table.

FUNCTION SUM (argument-1 argument-2 ...)

The SUM function returns an integer or numeric value that is the sum of the arguments.

The function type will be integer if all of the arguments are integers. The function type will be numeric if all of the arguments are numeric or if the arguments are mixed (some integer and some numeric).

Example:
```
COMPUTE TOTAL-AMOUNT = FUNCTION SUM(VAR-1 VAR-2 VAR-3).
```

The data item TOTAL-AMOUNT will contain the sum of the three variables.

The subscript ALL may be used to sum up a series of numbers in a table.

FUNCTION RANDOM (argument-1)

The RANDOM function returns a numeric value that is a pseudo-random number. The same argument value will always produce the same random number. The returned value will range between zero and one.

Argument-1 is optional. If the first reference to this function omits argument-1, a seed value of zero is used. Subsequent references to the function without argument-1 return the next number in the current sequence.
Example: `COMPUTE RANDOM-NUMBER = FUNCTION RANDOM(20).`

The data item `RANDOM-NUMBER` will contain a number that the function calculated using the number 20 as a seed value.

FUNCTION REVERSE(argument-1)

The `REVERSE` function returns a character string of exactly the same length as the argument, and containing the same characters as the argument, but in reverse order.

The function type is alphanumeric.

Argument-1 may be alphabetic or alphanumeric and must be at least one character in length.

Example: `MOVE FUNCTION REVERSE(CUST-NAME) TO REVERSE-NAME.`

If `CUST-NAME` contains `JOHNSONbbb`, then `REVERSE-NAME` will contain `bbbNOSNHOJ`.

FUNCTION UPPER-CASE(argument-1)

The `UPPER-CASE` function returns a character string that is the same length as the argument, with each lower-case letter replaced by the corresponding upper-case letter.

The function type is alphanumeric.

The argument must be either alphabetic or alphanumeric and must be at least one character in length.

Example: `MOVE FUNCTION UPPER-CASE(VAR-1) TO VAR-2.`

If `VAR-1` contains the value 'abcde', then `VAR-2` will contain '`ABCDE`'.

MORE INTRINSIC DATE FUNCTIONS

Cobol has several intrinsic functions pertaining to dates. These functions involve 4-digit years and are Y2K compliant.

The following material covers the functions `INTEGER-OF-DATE`, `DATE-OF-INTEGER`, and `CURRENT-DATE`. It does not cover the related Julian date functions `INTEGER-OF-DAY` and `DAY-OF-INTEGER`.

Cobol's intrinsic date functions number the days following December 31, 1600. Thus, January 1, 1601 has the integer value 1. January 1, 2000 has the integer value 145732.

FUNCTION INTEGER-OF-DATE(argument-1)

Argument: An elementary numeric data item containing a date in the format yyyymmdd

Returns: An integer that is at least 6 digits long. A field intended to hold the returned value must be an elementary numeric data item that can hold at least 6 digits.

FUNCTION DATE-OF-INTEGER(argument-1)

Argument: An elementary numeric data item containing an integer representing the number of days since December 31, 1600.

Returns: A date in the format yyyymmdd. A field intended to hold the returned value must be an elementary numeric data item that is 8 digits long.

Examples of Date Arithmetic Using the Functions

1. Suppose we want to find the number of days between March 15, 1999, and and October 30, 1999. We can use the `INTEGER-OF-DATE` function to turn each date into an integer, and then subtract the integers.

   In WORKING-STORAGE:

```
01  START-DATE      PIC 9(8)  VALUE 19990315.
01  END-DATE        PIC 9(8)  VALUE 19991030.

01  START-INT       PIC 9(6).
01  END-INT         PIC 9(6).

01  DAYS-BETWEEN    PIC 9(6).
```

   In the `PROCEDURE DIVISION`:

```
COMPUTE START-INT = FUNCTION INTEGER-OF-DATE (START-DATE).
COMPUTE END-INT = FUNCTION INTEGER-OF-DATE (END-DATE).
COMPUTE DAYS-BETWEEN = END-INT - START-INT.
```

   If the following `DISPLAY`s were included:

```
DISPLAY 'START-INT= ' START-INT.
DISPLAY 'END-INT= ' END-INT.
DISPLAY 'DAYS-BETWEEN= ' DAYS-BETWEEN.
```

   The output from the `DISPLAY`s would look like this:

```
START-INT= 145440
END-INT= 145669
DAYS-BETWEEN= 000229
```

   In place of the three statements, a single statement could be used:

```
COMPUTE DAYS-BETWEEN =
             FUNCTION INTEGER-OF-DATE (END-DATE) -
             FUNCTION INTEGER-OF-DATE (START-DATE).
```

This is a a single compute statement that subtracts one returned integer from another eliminates the need for the working storage variables `START-INT` and `END-INT`.

2. Suppose we want to find the number of days between March 15, 1999 and the current date (whatever that is).

   In working storage, change the definition of `END-DATE`:

```
01  TODAY.
    05  END-DATE       PIC 9(8).
```

   Or, alternatively:

```
01  TODAY              PIC X(8).
01  END-DATE           REDEFINES TODAY
                       PIC 9(8).
```

   Note that either of the above results in an *alphanumeric* field (`TODAY`), which is a requirement of the `CURRENT-DATE` function.

   In working storage, get the end date before subtracting:

```
MOVE FUNCTION CURRENT-DATE TO TODAY.

COMPUTE DAYS-BETWEEN =
             FUNCTION INTEGER-OF-DATE (END-DATE) -
             FUNCTION INTEGER-OF-DATE (START-DATE).
```

3. Suppose we want to find out what the date was 100 days ago.

```
MOVE FUNCTION CURRENT-DATE TO TODAY.

COMPUTE END-INT = FUNCTION INTEGER-OF-DATE (END-DATE).
COMPUTE START-INT = START-INT - 100.
COMPUTE START-DATE = FUNCTION DATE-OF-INTEGER (START-INT).
```

4.11 COBOL Control Break Processing

In control break processing, we have an input file which has been sorted on several key fields. We can think of the group as occurring in large groups of records, each group having the same value of the first key. Within each large group, the records occur in smaller groups according to the value of the second key. This pattern continues, depending on the number of keys.

We are interested in statistics (sum, maximum, minimum, etc.) for the file as a whole, for each large group, for each small group within each large group, etc.

Pseudocode

```
Main routine:
    Zero grand totals
    Read first record
    Perform Process-One-Large-Group
      until end of file
    Print grand totals

Process-One-Large-Group routine:
    Save large group identifier
    Zero large group subtotals
    Fill in headings with identifier for large group
    Perform Process-Smaller-Group
      until new identifier for large group
        or end of file
    Print subtotals for large group
    Add large group subtotals to grand totals

Process-Smaller-Group routine:
    Save smaller group identifier
    Zero smaller group subtotals
    Fill in headings with identifier for smaller group
    Perform Process-Smallest-Group
      until new identifier for smaller group
        or new identifier for large group
        or end of file
    Print subtotals for smaller group
    Add smaller group subtotals to large group subtotals

Process-Smallest-Group routine:
    Save smallest group identifier
    Zero smallest group subtotals
    Fill in headings with identifier for smallest group
    Perform Process-One-Record
      until new identifier for smallest group
        or new identifier for smaller group
        or new identifier for large group
        or end of file
    Print subtotals for smallest group
```

Add smallest group subtotals to smaller group subtotals

Process-One-Record routine:
Do whatever needs to be done with the record
(depends on the purpose of the program)
Read the next record

Notes

The above pseudocode assumes we are adding up totals. We could easily instead be looking for a maximum value or a minimum value at each point. There may be a lot of variables involved.

If we have more than 3 levels of control, there can be any number of routines between Main and Process-One-Record, but all of them look very similar. If we have only 1 key, control-break processing loses its distinctive meaning.

The priming Read is in Main; the other Read is in Process-One-Record.

All of this is absolutely dependent on the data being sorted in order by the various key fields, which are the identifiers for the large group, smaller group, and smallest group, in that order. If we want other statistics, we need to re-sort the data and write a new program.

There is no assumption here that each large group contains the same number of smaller groups, or that each smaller group contains the same number of smallest groups. If that was the case, we could theoretically put all the data in a multidimensional table--if we could have a table large enough.

Example

Suppose our records look like this:

```
Salesperson-Name     City     Month     Sales-Total   Sales-Data
```

The data is sorted on 3 keys: 1st Salesperson-Name, 2nd City, 3rd Month.
We want a report like this:

```
Name:  John Jones        City:  Baltimore     Month:  June

     Sales-Data                   Sales-Amount
     ..........                     $.......
     ..........                     $.......
     ..........                     $.......
     ..........                     $.......

                          Total = $........
```

Each time a control field (a key field) changes, we have a control break. We advance to a new page. (We may end up with a lot of pages of output.) For control breaks higher than the first, we will also print totals for the city, for the salesperson, or (last) for the company as a whole.

Here is pseudocode for this example:

```
000-Main.

    Initialize company total.
    Priming read.
    Perform 100-Process-One-Salesperson until End-of-File.
    Print company total.

100-Process-One-Salesperson.

    Save the salesperson name and put it in the heading.
    Initialize salesperson total.
    Perform 200-Process-One-City until End-of-File or
        we have a new salesperson.
    Add salesperson total to company total.
    Print salesperson total.

200-Process-One-City.

    Save the city name and put it in the heading.
    Initialize city total.
    Perform 300-Process-One-Month until End-of-File or
        we have a new salesperson or
        we have a new city.
    Add city total to salesperson total.
    Print city total.

300-Process-One-Month.
    Save the month name and put it in the heading.
    Advance to a new page and print the heading.
    Initialize month total.
    Perform 400-Process-One-Sale until End-of-File or
        we have a new salesperson or
        we have a new city or
        we have a new month.
    Add month total to city total.
    Print month total.

400-Process-One-Sale.

    Print a line about the latest sale.
    Add sales amount to month total.
    Read next record.
```

4.10 Example COBOL Data Validation Program

```
 IDENTIFICATION DIVISION.

*************************************************************
* PROGRAM NAME:  DATAVAL                                    *
*                                                           *
* AUTHORS:       programmer name                            *
*                                                           *
* FUNCTION:      This program validates input               *
*                transactions that will be used by          *
*                another program to update a                *
*                a sequential master file of patient        *
*                records.  A separate error message is      *
*                produced for each error found on an        *
*                invalid record.                            *
*                                                           *
* INPUT:         A file, on disk, containing a header       *
*                record with the date the file was          *
*                created, followed by an unspecified        *
*                number of Add, Alter, and Delete           *
*                transactions.                              *
*                                                           *
* OUTPUT:        (1) A file, on disk, containing the        *
*                valid records, and (2) an error report     *
*                showing each invalid record, along         *
*                with one or more error messages.           *
*                                                           *
* NOTES:         NONE                                       *
*************************************************************

 PROGRAM-ID.  DATAVAL.

 ENVIRONMENT DIVISION.

 INPUT-OUTPUT SECTION.

 FILE-CONTROL.

     SELECT TRAN-FILE   ASSIGN TO OLDTRAN.
     SELECT NEW-FILE    ASSIGN TO NEWTRAN.
     SELECT PRINT-FILE  ASSIGN TO PRINTER.

 DATA DIVISION.

 FILE SECTION.
*************************************************************
* The input file contains the transactions to be            *
* validated.                                                *
* The transactions are preceded by one header record.       *
*************************************************************
```

```
FD  TRAN-FILE
    RECORD CONTAINS 83 CHARACTERS
    RECORDING MODE IS F.

01  HEADER-RECORD.
    05  DATE-BLANKS              PIC X(8).
    05  DATE-FIELD               PIC 9(8).
    05  DATE-MORE-BLANKS         PIC X(67).

01  TRAN-RECORD.
    05  TRAN-PATIENT-NUM         PIC 9(5).
    05  TRAN-CODE                PIC 9.
    05  TRAN-INFO                PIC X(77).
    05  TRAN-ADD-INFO REDEFINES TRAN-INFO.
        10  TRAN-ADD-PATIENT     PIC X(20).
        10  TRAN-ADD-DOCTOR      PIC X(10).
        10  TRAN-ADD-LAST-VISIT  PIC 9(8).
        10  TRAN-ADD-DX.
            15  TRAN-ADD-DXI     PIC X.
            15  TRAN-ADD-DXII    PIC 99.
        10  TRAN-ADD-NEXT-VISIT  PIC 9(8).
        10  TRAN-ADD-CHARGE      PIC 9999V99.
        10  TRAN-ADD-PT-PAID     PIC 9999V99.
        10  TRAN-ADD-INS-CO      PIC X(10).
        10  TRAN-ADD-INS-PAID    PIC 9999V99.
    05  TRAN-ALTER-INFO REDEFINES TRAN-INFO.
        10  TRAN-ALTER-CODE      PIC 9.
        10  TRAN-ALTER-FIELD     PIC X(76).
        10  TRAN-ALTER-PATIENT REDEFINES TRAN-ALTER-FIELD.
            15  TRAN-NEW-NAME    PIC X(20).
            15  FILLER           PIC X(56).
        10  TRAN-ALTER-DOCTOR REDEFINES TRAN-ALTER-FIELD.
            15  TRAN-NEW-DOCTOR  PIC X(10).
            15  FILLER           PIC X(66).
        10  TRAN-ALTER-LAST-VISIT REDEFINES
                                     TRAN-ALTER-FIELD.
            15  TRAN-NEW-LAST-VISIT  PIC 9(8).
            15  FILLER           PIC X(68).
        10  TRAN-ALTER-DX REDEFINES TRAN-ALTER-FIELD.
            15  TRAN-NEW-DXI     PIC X.
            15  TRAN-NEW-DXII    PIC 99.
            15  FILLER           PIC X(73).
        10  TRAN-ALTER-NEXT-VISIT REDEFINES
                                     TRAN-ALTER-FIELD.
            15  TRAN-NEW-NEXT-VISIT   PIC 9(8).
            15  FILLER           PIC X(68).
        10  TRAN-ALTER-CHARGE REDEFINES TRAN-ALTER-FIELD.
            15  TRAN-NEW-CHARGE  PIC 9999V99.
            15  FILLER           PIC X(70).
        10  TRAN-ALTER-PAID REDEFINES TRAN-ALTER-FIELD.
```

```
            15  TRAN-NEW-PT-PAID PIC 9999V99.
            15  FILLER           PIC X(70).
        10  TRAN-ALTER-INS-CO REDEFINES TRAN-ALTER-FIELD.
            15  TRAN-NEW-INS-CO  PIC X(10).
            15  FILLER           PIC X(66).
        10  TRAN-ALTER-INS-PAID REDEFINES
                                      TRAN-ALTER-FIELD.
            15  TRAN-NEW-INS-PAID  PIC 9999V99.
            15  FILLER           PIC X(70).
    05  TRAN-DELETE-INFO REDEFINES TRAN-INFO.
        10  TRAN-DELETE-BLANKS   PIC X(77).

*********************************************************
* This output file contains a generic print line that   *
* is used to print all of the lines on the report.      *
*********************************************************

 FD  PRINT-FILE
     LABEL RECORDS ARE OMITTED
     RECORDING MODE IS F.

 01  PRINT-LINE                  PIC X(132).

*********************************************************
* This output file, on disk, contains all of the valid  *
* records from the input file.                          *
*********************************************************

 FD  NEW-FILE
     LABEL RECORDS ARE STANDARD
     BLOCK CONTAINS 30 RECORDS
     RECORDING MODE IS F.

 01  NEW-RECORD                PIC X(83).


 WORKING-STORAGE SECTION.

*********************************************************
* VARIABLE DICTIONARY                                   *
*                                                       *
* SYS-DATE:  Holds date accepted from system.           *
* SYS-TIME:  Holds time accepted from system.           *
*                                                       *
* NEW-FILE-HEADER:  Header record for the edited tran   *
*                   file.                               *
*                                                       *
* TITLE-LINE1:  Page headers.                           *
* TITLE-LINE2:                                          *
*                                                       *
* COLUMN-LINE1:         Column headers.                 *
```

```
* COLUMN-LINE2:                                          *
*                                                       *
* SUMMARY-TITLE-LINE2:  Page header for the contents of  *
*                       the valid file.                 *
*                                                       *
* DETAIL-LINE:          Prints the contents of the       *
*                       record above the error messages. *
*                                                       *
* FIRST-ERROR-LINE:     Prints the first error line.     *
*                                                       *
* NEXT-ERROR-LINE:      Prints any subsequent error      *
*                       lines for the same transaction.  *
*                                                       *
* SUMMARY-LINE:         Prints the number of valid and   *
*                       invalid records processed.       *
*                                                       *
* NEW-RECORD-LINE:      For printing contents of valid   *
*                       file.                           *
*                                                       *
* EOF-FLAG:             Set to 'y' when end of file      *
*                       occurs.                         *
*                                                       *
* HEADER-ERROR-FLAG:    Set to 'y' if header on input    *
*                       file is not valid.              *
*                                                       *
* ERROR-FLAG:           Set to 'y' when the first error  *
*                       for a transaction is printed.   *
*                                                       *
* ERROR-SUB:            Subscript for error message      *
*                       table.                          *
*                                                       *
* ERROR-TOTAL           Count of invalid transactions.   *
*                                                       *
* VALID-TOTAL           Count of valid transactions.     *
*                                                       *
* COUNT-LINE            Line counter.                    *
*                                                       *
* COUNT-PAGE            Page counter.                    *
*                                                       *
* ERROR-TABLE           Table of error messages.         *
*********************************************************

 01  SYSTEM-DATE-AND-TIME.
     05  SYS-DATE           PIC 9(8).
     05  SYS-TIME           PIC 9(8).

 01  NEW-FILE-HEADER.
     05                     PIC X(8).
     05                     PIC X(8).
     05  PROGRAMMER-NAME    PIC X(23).
     05                     PIC X(44).
```

```
01  TITLE-LINE1.
    05                    PIC X(5)   VALUE SPACES.
    05  TITLE-DATE        PIC 99/99/9999.
    05                    PIC X(22)  VALUE SPACES.
    05                    PIC X(26)
        VALUE 'COUNTRYSIDE MEDICAL CLINIC'.
    05                    PIC X(21)  VALUE SPACES.
    05                    PIC X(5)   VALUE 'PAGE '.
    05  LINE-COUNT-PAGE   PIC Z9.

01  TITLE-LINE2.
    05                    PIC X(34)  VALUE SPACES.
    05                    PIC X(32)
        VALUE 'DATA VALIDATION EXCEPTION REPORT'.
    05                    PIC X(50)  VALUE SPACES.

01  COLUMN-LINE1.
    05                  PIC X(28)
                        VALUE 'PATIENT TRAN     1         2'.
    05                  PIC X(20)
                        VALUE '         3         4'.
    05                  PIC X(20)
                        VALUE '         5         6'
    05                  PIC X(20)
                        VALUE '         7         8'.
01  COLUMN-LINE2.
    05                    PIC X(21)
        VALUE 'NUMBER  CODE  7890123'.
    05                    PIC X(70)
        VALUE ALL '4567890123'.

01  SUMMARY-TITLE-LINE2.
    05                    PIC X(40)   VALUE SPACES.
    05                    PIC X(20)
        VALUE 'VALIDATED DATA ITEMS'.
    05                    PIC X(56)   VALUE SPACES.

01  DETAIL-LINE.
    05  DET-PATIENT-NUM   PIC X(5).
    05                    PIC X(5)    VALUE SPACES.
    05  DET-TRAN-CODE     PIC X.
    05                    PIC XXX     VALUE SPACES.
    05  DET-CONTENT            PIC X(113).

01  FIRST-ERROR-LINE.
    05                    PIC X(19)   VALUE SPACES.
    05                    PIC X(20)
        VALUE 'ERROR MESSAGES:     '.
    05  FIRST-ERROR-NAME  PIC X(44).
    05                    PIC X(49)   VALUE SPACES.
01  NEXT-ERROR-LINE.
```

```
    05                      PIC X(39)   VALUE SPACES.
    05  NEXT-ERROR-NAME     PIC X(44).
    05                      PIC X(49)   VALUE SPACES.

01  SUMMARY-LINE.
    05                      PIC X(20).
    05                      PIC X(7)    VALUE 'TOTALS:'.
    05                      PIC X(5)    VALUE SPACES.
    05                      PIC X(16)
        VALUE 'INVALID RECORDS='.
    05  LINE-ERROR-TOTAL    PIC ZZ9.
    05                      PIC X(5)    VALUE SPACES.
    05                      PIC X(14)
        VALUE 'VALID RECORDS='.
    05  LINE-VALID-TOTAL    PIC ZZ9.
    05                      PIC X(59).

01  NEW-RECORD-LINE.
    05                      PIC X(5)    VALUE SPACES.
    05  NEW-RECORD-STUFF    PIC X(83).
    05                      PIC X(44)   VALUE SPACES.

01  FLAGS-SUBS.
    05  EOF-FLAG            PIC X       VALUE 'N'.
    05  HEADER-ERROR-FLAG   PIC X       VALUE 'N'.
        88  NO-HEADER-ERROR             VALUE 'N'.
    05  ERROR-FLAG          PIC X       VALUE 'N'.
    05  ERROR-SUB           PIC 99.

01  TOTALS-COUNTS.
    05  ERROR-TOTAL         PIC 999     VALUE 0.
    05  VALID-TOTAL         PIC 999     VALUE 0.
    05  COUNT-LINE          PIC 99      VALUE 0.
    05  COUNT-PAGE          PIC 99      VALUE 0.

01  ERROR-MESSAGE.
    05                      PIC X(44)
        VALUE IS 'PATIENT NUMBER IS NOT NUMERIC'.
    05                      PIC X(44)
        VALUE IS 'PATIENT NUMBER IS ZERO'.
    05                      PIC X(44)
        VALUE IS 'TRAN-CODE IS NOT NUMERIC'.
    05                      PIC X(44)
        VALUE IS 'TRAN-CODE VALUE IS OUT OF VALID RANGE'.
    05                      PIC X(44)
        VALUE IS 'ALTER CODE IS NOT NUMERIC'.
    05                      PIC X(44)
        VALUE IS 'ALTER-CODE VALUE IS OUT OF VALID RANGE'.
    05                      PIC X(44)
        VALUE IS 'PATIENT NAME MISSING - BLANKS FOUND'.
    05                      PIC X(44)
```

```
           VALUE IS 'PHYSICIAN NAME MISSING - BLANKS FOUND'.
       05                      PIC X(44)
           VALUE IS 'DATE OF LAST VISIT IS NOT NUMERIC'.
       05                      PIC X(44)
           VALUE IS 'DATE OF LAST VISIT IS ZERO'.
       05                      PIC X(44)
           VALUE IS 'DIAGNOSIS CODE PART A IS INVALID VALUE'.
       05                      PIC X(44)
           VALUE IS 'DIAGNOSIS CODE PART B IS NOT NUMERIC'.
       05                      PIC X(44)
           VALUE IS 'DATE OF NEXT VISIT IS NOT NUMERIC'.
       05                      PIC X(44)
           VALUE IS 'NEW CHARGE IS NOT NUMERIC'.
       05                      PIC X(44)
           VALUE IS 'AMOUNT PAID BY PATIENT IS NOT NUMERIC'.
       05                      PIC X(44)
           VALUE IS 'AMOUNT PAID BY INSURANCE NOT NUMERIC'.
       05                      PIC X(44)
           VALUE IS 'DELETE RECORD IS NOT ALL BLANKS'.

    01  ERROR-TABLE REDEFINES ERROR-MESSAGE.
        05  ERROR-NAME         PIC X(44) OCCURS 17.

    PROCEDURE DIVISION.

   ********************************************************
   * 000-MAIN performs a subroutine to check the header on  *
   * the input file.  If OK, it then writes a header on the *
   * valid output file, gets the system date and time,      *
   * writes headers on the error report, does a priming     *
   * read for the first transaction record, and performs a  *
   * subroutine to process transaction records until end of *
   * file.  Finally, it prints the total number of valid    *
   * and invalid records, and then performs a subroutine to *
   * print out the contents of the valid transaction file.  *
   ********************************************************

    0000-MAIN.

        OPEN INPUT TRAN-FILE.
        OPEN OUTPUT PRINT-FILE
                    NEW-FILE.

        PERFORM 0100-CHECK-HEADER.

        IF NO-HEADER-ERROR
            MOVE HEADER-RECORD TO NEW-FILE-HEADER
            MOVE 'LYNN BONNETT' TO PROGRAMMER-NAME
            WRITE NEW-RECORD FROM NEW-FILE-HEADER
            MOVE DATE-FIELD TO TITLE-DATE
```

```
           MOVE FUNCTION CURRENT-DATE(1:16)TO
                SYSTEM-DATE-AND-TIME

           READ TRAN-FILE
               AT END MOVE 'Y' TO EOF-FLAG
           END-READ

           PERFORM 0200-PROCESS-RECORD UNTIL EOF-FLAG = 'Y'

           MOVE ERROR-TOTAL TO LINE-ERROR-TOTAL
           MOVE VALID-TOTAL TO LINE-VALID-TOTAL
           WRITE PRINT-LINE FROM SUMMARY-LINE AFTER 4

           CLOSE NEW-FILE
           OPEN INPUT NEW-FILE
           PERFORM 0300-PRINT-NEW-FILE

       END-IF.

       CLOSE TRAN-FILE
             PRINT-FILE
             NEW-FILE.

       GOBACK.

 0000-EXIT. EXIT.

*************************************************************
* 0100-CHECK-HEADER reads the first record in the input  *
* file, makes sure it is present, and that it is a valid *
* header record.  An error message is displayed if it is *
* not valid.                                             *
*************************************************************

 0100-CHECK-HEADER.

       READ TRAN-FILE
           AT END MOVE 'Y' TO EOF-FLAG
       END-READ.

       IF EOF-FLAG = 'Y'
           DISPLAY 'INPUT TRAN FILE IS EMPTY'
           MOVE 'Y' TO HEADER-ERROR-FLAG
       ELSE
           IF DATE-BLANKS IS NOT EQUAL TO SPACES
               DISPLAY 'HEADER KEY FIELD IS INVALID'
               MOVE 'Y' TO HEADER-ERROR-FLAG
           ELSE
               IF DATE-FIELD IS NOT NUMERIC
                   DISPLAY 'HEADER DATE IS INVALID'
                   MOVE 'Y' TO HEADER-ERROR-FLAG
```

```
            ELSE
                IF DATE-MORE-BLANKS NOT EQUAL TO SPACES
                    DISPLAY 'UNEXPECTED DATA IN HEADER'
                    MOVE 'Y' TO HEADER-ERROR-FLAG
                END-IF
            END-IF
        END-IF
    END-IF.

 0100-EXIT. EXIT.

*************************************************************
* 0200-PROCESS-RECORD validates the patient and the      *
* transaction code.  If the transaction code is valid,   *
* it performs either 0220-TRAN-ADD or 0230-TRAN-ALTER or *
* 0240-TRAN-DELETE to validate the rest of the record.   *
* Then, if no error was found on the transaction, it     *
* writes the transaction to the valid file and           *
* increments the count of valid records.  Finally, it    *
* reads the next transaction record.                     *
*************************************************************

 0200-PROCESS-RECORD.

    MOVE 'N' TO ERROR-FLAG.

    IF TRAN-PATIENT-NUM IS NOT NUMERIC
      MOVE 1 TO ERROR-SUB
      PERFORM 0210-PRINT-ERROR
    ELSE
      IF TRAN-PATIENT-NUM IS ZERO
         MOVE 2 TO ERROR-SUB
         PERFORM 0210-PRINT-ERROR
      END-IF
    END-IF.

    IF TRAN-CODE IS NOT NUMERIC
        MOVE 3 TO ERROR-SUB
        PERFORM 0210-PRINT-ERROR
    ELSE
        EVALUATE TRAN-CODE
            WHEN 1
                PERFORM 0220-TRAN-ADD
            WHEN 2
                PERFORM 0230-TRAN-ALTER
            WHEN 3
                PERFORM 0240-TRAN-DELETE
            WHEN OTHER
               MOVE 4 TO ERROR-SUB
               PERFORM 0210-PRINT-ERROR
         END-EVALUATE
```

```
     END-IF.

     IF ERROR-FLAG = 'N'
          ADD 1 TO VALID-TOTAL
          WRITE NEW-RECORD FROM TRAN-RECORD
     END-IF.

     READ TRAN-FILE
          AT END MOVE 'Y' TO EOF-FLAG
     END-READ.

 0200-EXIT. EXIT.

*************************************************************
* 0210-PRINT-ERROR checks the error flag to see if this  *
* is the first error for this transaction ("N" indicates *
* first error).  For a first error, this routine checks  *
* whether page and column heads are needed (performing   *
* 0211-PRINT-TITLE to print them if needed).  It adds to *
* the count of invalid records.  Then it prints two      *
* lines (one with the transaction data, and one with the *
* error message preceded by the words "ERROR MESSAGES:". *
* If not the first, it prints only the error message.    *
*************************************************************

 0210-PRINT-ERROR.

     IF ERROR-FLAG = 'N'
         IF COUNT-LINE >= 30
             PERFORM 0211-PRINT-TITLE
         END-IF
         ADD 1 TO ERROR-TOTAL

         MOVE TRAN-PATIENT-NUM TO DET-PATIENT-NUM
         MOVE TRAN-CODE TO DET-TRAN-CODE
         MOVE TRAN-INFO TO DET-CONTENT
         WRITE PRINT-LINE FROM DETAIL-LINE AFTER 2

         MOVE ERROR-NAME (ERROR-SUB) TO FIRST-ERROR-NAME
         WRITE PRINT-LINE FROM FIRST-ERROR-LINE
         MOVE 'Y' TO ERROR-FLAG
         ADD 2 TO COUNT-LINE
     ELSE
         MOVE ERROR-NAME (ERROR-SUB) TO NEXT-ERROR-NAME
         WRITE PRINT-LINE FROM NEXT-ERROR-LINE
         ADD 1 TO COUNT-LINE
     END-IF.

 0210-EXIT. EXIT.

*************************************************************
```

```
* 0220-TRAN-ADD checks each field on an add transaction. *
* Whenever an error is found, it moves the appropriate   *
* subscript to ERROR-SUB, and performs 0210-PRINT-ERROR  *
* to process the error.                                  *
**********************************************************

 0220-TRAN-ADD.

     IF TRAN-ADD-PATIENT IS = SPACES
        MOVE 7 TO ERROR-SUB
        PERFORM 0210-PRINT-ERROR
     END-IF.

     IF TRAN-ADD-DOCTOR IS = SPACES
        MOVE 8 TO ERROR-SUB
        PERFORM 0210-PRINT-ERROR
     END-IF.

     IF TRAN-ADD-LAST-VISIT IS NOT NUMERIC
        MOVE 9 TO ERROR-SUB
        PERFORM 0210-PRINT-ERROR
     ELSE
        IF TRAN-ADD-LAST-VISIT IS ZERO
            MOVE 10 TO ERROR-SUB
            PERFORM 0210-PRINT-ERROR
        END-IF
     END-IF.

     IF TRAN-ADD-DXI IS NOT =
                'A' AND 'C' AND 'N' AND 'R' AND 'Y'
        MOVE 11 TO ERROR-SUB
        PERFORM 0210-PRINT-ERROR
     END-IF.

     IF TRAN-ADD-DXII IS NOT NUMERIC
        MOVE 12 TO ERROR-SUB
        PERFORM 0210-PRINT-ERROR
     END-IF.

     IF TRAN-ADD-NEXT-VISIT IS NOT NUMERIC
        MOVE 13 TO ERROR-SUB
        PERFORM 0210-PRINT-ERROR
     END-IF.

     IF TRAN-ADD-CHARGE IS NOT NUMERIC
        MOVE 14 TO ERROR-SUB
        PERFORM 0210-PRINT-ERROR
     END-IF.

     IF TRAN-ADD-PT-PAID IS NOT NUMERIC
        MOVE 15 TO ERROR-SUB
```

```
           PERFORM 0210-PRINT-ERROR
        END-IF.

        IF TRAN-ADD-INS-PAID IS NOT NUMERIC
              MOVE 16 TO ERROR-SUB
              PERFORM 0210-PRINT-ERROR
        END-IF.

    0220-EXIT. EXIT.

   ***********************************************************
   * 0230-TRAN-ALTER validates the alter code.  For each     *
   * possible valid alter code, it checks the appropriate    *
   * field.  If the field is not valid, it moves the         *
   * appropriate subscript to ERROR-SUB and performs 500-    *
   * PRINT-ERROR to process the error.                       *
   ***********************************************************

    0230-TRAN-ALTER.

        IF TRAN-ALTER-CODE IS NOT NUMERIC
              MOVE 5 TO ERROR-SUB
              PERFORM 0210-PRINT-ERROR
        ELSE
          EVALUATE TRAN-ALTER-CODE
           WHEN 1
              IF TRAN-NEW-NAME IS = SPACES
                   MOVE 7 TO ERROR-SUB
                   PERFORM 0210-PRINT-ERROR
              END-IF
           WHEN 2
              IF TRAN-NEW-DOCTOR IS = SPACES
                 MOVE 8 TO ERROR-SUB
                 PERFORM 0210-PRINT-ERROR
              END-IF
           WHEN 3
              IF TRAN-NEW-LAST-VISIT IS NOT NUMERIC
                 MOVE 9 TO ERROR-SUB
                 PERFORM 0210-PRINT-ERROR
              ELSE
                 IF TRAN-NEW-LAST-VISIT = ZEROES
                    MOVE 10 TO ERROR-SUB
                    PERFORM 0210-PRINT-ERROR
                 END-IF
              END-IF
           WHEN 4
              IF TRAN-NEW-DXI IS NOT = 'A' AND 'C' AND 'N'
                 AND 'R' AND 'Y'
                 MOVE 11 TO ERROR-SUB
                 PERFORM 0210-PRINT-ERROR
              END-IF
```

```
            IF TRAN-NEW-DXII IS NOT NUMERIC
               MOVE 12 TO ERROR-SUB
               PERFORM 0210-PRINT-ERROR
            END-IF
         WHEN 5
            IF TRAN-NEW-NEXT-VISIT IS NOT NUMERIC
               MOVE 13 TO ERROR-SUB
               PERFORM 0210-PRINT-ERROR
            END-IF
         WHEN 6
            IF TRAN-NEW-CHARGE IS NOT NUMERIC
                MOVE 14 TO ERROR-SUB
                PERFORM 0210-PRINT-ERROR
            END-IF
         WHEN 7
            IF TRAN-NEW-PT-PAID IS NOT NUMERIC
                MOVE 15 TO ERROR-SUB
                PERFORM 0210-PRINT-ERROR
            END-IF
         WHEN 8
            CONTINUE
         WHEN 9
            IF TRAN-NEW-INS-PAID IS NOT NUMERIC
                MOVE 16 TO ERROR-SUB
                PERFORM 0210-PRINT-ERROR
            END-IF
         WHEN OTHER
           MOVE 6 TO ERROR-SUB
           PERFORM 0210-PRINT-ERROR
         END-EVALUATE
     END-IF.
 0230-EXIT. EXIT.

*************************************************************
* 0240-TRAN-DELETE checks the rest of the delete         *
* transaction.  If the rest of the record is not all     *
* spaces, it places the appropriate subscript in         *
* ERROR-SUB and performs 0210-PRINT-ERROR to process the *
* error.                                                 *
*************************************************************

 0240-TRAN-DELETE.

     IF TRAN-DELETE-BLANKS IS NOT EQUAL TO SPACES
        MOVE 17 TO ERROR-SUB
        PERFORM 0210-PRINT-ERROR
     END-IF.

 0240-EXIT. EXIT.

*************************************************************
```

```
* 0211-PRINT-TITLE prints page and column headers, adds 1 *
* to the page counter and resets the line counter.        *
***********************************************************

 0211-PRINT-TITLE.

     WRITE PRINT-LINE FROM SYSTEM-DATE-AND-TIME AFTER PAGE.
     ADD 1 TO COUNT-PAGE.
     MOVE 0 TO COUNT-LINE.
     MOVE COUNT-PAGE TO LINE-COUNT-PAGE.
     WRITE PRINT-LINE FROM TITLE-LINE1 AFTER 2.
     WRITE PRINT-LINE FROM TITLE-LINE2.
     WRITE PRINT-LINE FROM COLUMN-LINE1 AFTER 2.
     WRITE PRINT-LINE FROM COLUMN-LINE2.

 0211-EXIT. EXIT.

*************************************************************
* 0300-PRINT-NEW-FILE prints out the contents of the new     *
* disk file containing the valid transactions.  It puts page *
* page and column heads on the first page of this output.    *
*************************************************************

 0300-PRINT-NEW-FILE.

     WRITE PRINT-LINE FROM SYSTEM-DATE-AND-TIME AFTER PAGE.
     ADD 1 TO COUNT-PAGE.
     MOVE COUNT-PAGE TO LINE-COUNT-PAGE.
     WRITE PRINT-LINE FROM TITLE-LINE1 AFTER 2.
     WRITE PRINT-LINE FROM SUMMARY-TITLE-LINE2.
     MOVE SPACES TO PRINT-LINE.
     WRITE PRINT-LINE.

     MOVE 'N' TO EOF-FLAG.

     READ NEW-FILE
       AT END MOVE 'Y' TO EOF-FLAG
     END-READ.

     PERFORM UNTIL EOF-FLAG = 'Y'
       MOVE NEW-RECORD TO NEW-RECORD-STUFF
       WRITE PRINT-LINE FROM NEW-RECORD-LINE

       READ NEW-FILE
         AT END MOVE 'Y' TO EOF-FLAG
       END-READ

     END-PERFORM.

 0300-EXIT. EXIT.
```