Discussed is a program design language—a form of pseudo-code—that has been developed and used to organize, teach, document, and develop software systems. An example of top-down program design illustrates the key steps in using the language: determining the requirements, abstracting the functions, expanding the functions, and verifying the functions.

Syntax and conventions of the language are given in an appendix.

Top-down development using a program design language

by P. Van Leer

The McDonnell Douglas Automation Company (MCAUTO™), which is a division of the McDonnell Douglas Corporation, provides data processing services to the other divisions within the corporation as well as nationwide commercial data-processing services. Each division of McDonnell Douglas has its own systems analysts for business applications. The systems analysts determine user requirements and develop appropriate application system designs. Specifications for the system designs usually include a prose overview, descriptions of input and output files, screen layouts, report layouts, and logic specifications. MCAUTO personnel may assist the divisional analysts with the specifications or may develop the system designs. In either case, MCAUTO personnel are responsible for refining the system design, program design as necessary, coding, testing, and writing documentation necessary for the operation and maintenance of the systems. Thus programmers in MCAUTO are presented with system specifications of varying levels of detail, in which any two systems may be defined at different levels of detail, especially with reference to program logic.

Early in 1973, MCAUTO established a pilot test task force to determine how and to what extent they could use various improved programming techniques that had been developed by that time. A result of that task force was the development and teaching of structured programming and other improved programming techniques. Most of the programming techniques that were known[1] in 1973 were tried simultaneously. The conclusion[2] was that the new techniques showed promise, but too

much had been attempted at one time. Accordingly, the instruction continued in 1974, concentrating on structured programming, top-down programming, structured walk throughs, and program design language, the latter being the subject of this paper. As a result of this further effort[3], use of these four techniques has become a stated policy at MCAUTO. This investigation into the other techniques continued in 1975, with Hierarchy plus Input Process Output (HIPO) also showing great promise as a system analyst tool.[4]

The logic specification standards that have been used at MCAUTO are roughly equivalent to detailed flowcharts, in which numbered English sentences are substituted for the various flowchart symbols. As detailed flowcharts were formerly used to create the required detail, so were logic specification standards. Although flowcharts and logic specification standards proved adequate for smaller and less complex applications, it was recognized in the early 1970s that more complex applications are correspondingly more difficult to describe and specify by the use of flowcharts. That increasing size and complexity of applications had gradually outgrown the capability and scope of earlier logic specifications was the primary condition that set the stage for the new technique of using a program design language.

Program design language

The program design language that is presented in this paper is a tool for designing programs in detail prior to coding. At MCAUTO, the program design language is used both as a language and as a program development methodology. The program design language is syntactically simple and supports structured control figures[5] tailored for PL/1 and COBOL. The syntax of the language is described in the Appendix. Top-down program development methodology and elements of stepwise refinement[6] and levels of abstraction are used with the program design languages.[7] The methodology is described in this paper. At MCAUTO, programmers use the program design language in conjunction with structured walkthroughs, top-down implementation,[5] and structured programming.[3,4] Although the value of the program design language has not been evaluated apart from the other techniques, the language is believed to be a major contributor to increased productivity.

The program design language, as a form of pseudocode, has the following characteristics:

- Notation is used to state program logic and function in an easy-to-read, top-to-bottom fashion.
- It is not a compilable language.

- It is an informal method of expressing structured programming logic.
- It is similar to a programming language (such as COBOL or PL/1), but is not bound by formal syntactical language rules.
- Conventions exist that pertain to the use of structured figures and indentation to aid in the visual perception of the logic.
- The primary purpose is to enable one to express ideas in natural English prose.
- The language permits concentration on logical solutions to problems, rather than the form and constraints within which the solutions must be stated.
- The language uses flowchart replacements, program documentation, and technical communication at all levels.
- Program design is expressed readably, and can be converted easily to executable code.

The program design language was initially used to teach structured programming to the programmers. As a teaching aid, the language helped the programmers make the transition to thinking in terms of a hierarchy of routines that consisted of basic structured figures.[4] When programmers started to implement application systems using flow charts and other earlier methods in which the programs were of the nonhierarchical and nonstructured type, the refining process included making hierarchical and structured program designs. Using the program design language rather than structured flowcharts or structuring the standard logic specifications proved to be the easiest way to improve the program design. Continued use and refinement of the language has established it as the medium of choice for either creating or refining a detailed program design. Although more experience with HIPO is needed, it presently appears that HIPO[4] may become the medium of choice for system design, and further become an excellent input for detailed program design. In time, HIPO may be as useful to analysts as the program design language is to programmers.

Top-down program design

Simplicity is a key attribute to the program design language syntax, conventions for which are given in the Appendix. In general, when the language is written according to the guidelines to be discussed in this paper, statements in the language are easy to transform into programs. More importantly, the simplicity frees the designer, who is usually a programmer, to concentrate on developing the detailed logic of a program. While the systematic application of the program design language facilitates program design, the language is not a simplistic means of doing the whole job of programming. Detailed program design is an iterative process, with the possibility that details discovered in the later

stages of design may lead to modifications in previous portions. Although experience in using the language and familiarity with the application may reduce the impact of such incidents, one should usually plan to complete a detailed design before starting to code a program. Since the program design language is easier to change (or rewrite) than actual code, cleaning up a program design in that language is usually more cost-effective than cleaning up program code. The primary objective in defining a procedure for the systematic application of the program design language is to provide a general scheme of things to be done during detailed program design.

The systematic application of the language is to apply the principles of top-down programming to the detailed program design function, which we term "top-down program design." This implies that the process of program design can be described as a hierarchy of discrete functions, which further implies that the work product (the program design) should be a hierarchy of discrete units that ease program implementation in a top-down manner.

According to program design language conventions, the discrete units in the case of program design are one-entry-one-exit routines (as in structured programming) that are no larger than one page. In most cases, all the detailed logic for a program does not fit on one page, a fact that leads to a squeezing down of detail into lower-level routines, and results in a number of hierarchically related routines. The syntax and conventions of the program design language promote a program design that meets the objectives of top-down programming. An example that shows the squeezing of detail into lower-level routines and the formation of hierarchically related routines is given in the following section.

Top-down program design example

The top-down design process may be regarded as having the following three distinct phases:

- Determining requirements.
- Abstracting functions.
- Expanding functions.

Obviously, the time and effort needed for each of these phases depends on the designer's experience and ability. Likewise, the particular way in which the functions are designed depends on the amount and organization of the source information. If the source data for a program design do not include completed file designs, report layouts, and user input definitions, then the application system design is not ready to be expanded into a detailed

program design. Moreover, a system design should include, as necessary, functions that the program should perform and any constraints on the program (such as field edits or sequences of calculations). Even after assuming that one has at least the minimum system-level specification for the program, there may still be wide variations in the level and volume of details and in the organization of those details. The optimum system specification is a hierarchy of user-oriented functions that includes only those details that are directly related to a user's requirements.

The establishment of practical guidelines for the optimal level of detail and organization for system-level specifications requires the active cooperation of both analysts and programmers. Whether done by analysts or programmers, the following three basic functions of detailed program design must still be performed: determine the requirements, abstract the functions, and expand the functions.

determining requirements

At the time of a detailed program design, the determining of program requirements consists primarily of studying the system specifications for the program. Any items that are vague, missing, undefined, or contradictory should be clarified before plunging into detailed program design. If the system specifications do not, at some point, provide a simple statement of user requirements, then write down such items as they become apparent. This point is crucial because the abstractive process should be in terms of the user's requirements. Likely sources are the definitions of output reports, files, screens, etc. The report specifications for a simple report generation program might yield the following functions:

- Accumulate total sales for each salesman.
- Accumulate total sales for each district.
- Accumulate total sales for all districts.

Examination of the input specification for the program might reveal the following constraints:

- The sales file has only one kind of record.
- Each sales record includes salesman name and number, and district number.
- Sales records are in order by salesman identification within each district.
- There may be several sales records for a salesman.

Additional constraints, such as "skip to new page after printing a district total" might be found.

If it is assumed that the specifications at the source specification level of detail do not express the user's requirements, the objec-

tive is to build a complete list at this level. It is not necessary to organize the list. Rather, one should concentrate on discovering all the functions that the user wants to be performed. Assuming this criterion, one might reasonably eliminate all the previously listed functions and constraints except the following:

- Accumulate total sales for each salesman.
- Accumulate total sales for each district.
- Accumulate total sales for all districts.
- Skip to new page after printing a district total.

At this point, a discussion with the analyst or user might be profitable. In any case, the requirements should be thoroughly understood, so that abstracting the functions—which is discussed in the following section—may be started.

abstracting functions

Abstracting the functions consists of discriminating between functions that are subfunctions and those that are main functions. To begin abstracting the functions, one first decides whether there is one function in the list that implies all the others. If there is none, then the programmer invents such a comprehensive function (i.e., he abstracts a general statement). For example, the report program function might be to "Summarize Sales," which implies that all the other sales functions are subfunctions. In that case, what are the relationships among the five functions on a main and subfunction basis? A good starting point for decision making is to organize the list by grouping all functions that have related inputs or outputs and by ranking each group in a most-general-to-most-detailed order. Since the report program has only one input file and one output report, grouping is not necessary. Ranking the sales functions yields the following general-to-detailed list:

1. Accumulate total sales for all districts.
2. Accumulate total sales for each district.
3. Skip to new page after printing district total.
4. Accumulate total sales for each salesman.

It appears that 2 and 3 are at the same functional level; that is, 1 implies 2 and 3 implies 4. This relationship suggests some minor reordering, which is brought out by the following list:

1. Accumulate total sales for all districts.
2. Accumulate total sales for each district.
4. Accumulate total sales for each salesman.
3. Skip to new page after printing district total.

Compare the new list with the report layout and note that there is a good match-up, especially if the basic functions are expand-

ed to designate the various totals that are to be printed as follows:

A1. Accumulate total sales for all districts.
B1. Accumulate total sales for each district.
C1. Accumulate total sales for each salesman.
C2. Print total sales for each salesman.
B2. Print total sales for each district.
B3. Skip to new page after printing district total.
A2. Print total sales for all districts.

At this point, the following three functional levels have been identified: all districts, each district, and each salesman. Each functional level contains a mixture of relatively simple functions, e.g., print and skip; and more general functions, e.g., accumulate. Generally, one cannot code a program from abstractions of function at this level. Definitions of the too-general functions must be expanded until all functions are sufficiently defined.

expanding functions

The expanding of functions consists of repeating the following four basic steps until all functions in the design have been sufficiently simplified to be coded: selection, analysis, specification, and verification. The appropriate point at which to stop depends on a programmer's familiarity with the program design language, structured programming, and the functions. Usually, the greater a programmer's experience with the program design language, the higher will be the level of detail that he uses. That is, when a programmer first starts using the language, more detailed definitions are needed (and written) than are needed after he has become accustomed to using the language. If a next lower level of expansion of named functions results in program design language statements that are program code, then the current level of expansion is probably sufficient. Of course, if all the statements can already be transferred into code on a one-for-one basis, the design is complete.

Selecting a function is the first step in expanding the functions. Expansion should generally be accomplished in a top-down manner. That is, expand the highest level (as yet undefined) function next. When faced with a choice of undefined functions at the same level, the main-line, or most important function, is usually expanded first. In the program example used in this paper, the function labeled A1 is the natural candidate for being expanded first. Since the expansion of A1 may produce another function that needs expansion, it is premature to assert that B1 should be expanded next. After having selected a function, the next step is to analyze it.

Analyzing a function is the process of deciding what must be done to accomplish a given function. This is sometimes referred

to as breaking a function down into subfunctions. In the event that major subfunctions have already been determined, analysis may consist of defining supportive subfunctions. For example, B1, B2, and B3 are major subfunctions of A1. Supportive subfunctions of A1 might be the following:

Set total for all districts to zero.
Add district sales to grand total.

Since A is the highest level in the program, the following data processing functions must also be done:

Open files.
Close files.

After the subfunctions have been identified, their relationships to one another can be specified.

Specifying relationships of the various subfunctions is accomplished by using the appropriate conditions and structured control figures. Specification may be done by using existing data variables, or it may require the definition of new data variables. New data variables should be noted as such, to facilitate both the eventual coding of a function and the expansion of lower-level functions during design. In effect, subfunctions and their relationships to one another should constitute a complete definition of function. For example, the A level might be specified as follows:

Summarize sales
Open files.
Set total for all districts to zero.
DO WHILE more sales data.
    Accumulate total for a district.
    Add total for district to total for all districts.
ENDDO
Print total sales for all districts.
Close files.

In this example, the statement "Accumulate total for a district" refers to the B- and C-level functions. We, therefore, proceed with the selection, analysis, and specification of the B- and C-level functions.

Accumulate total for a district
Set total for a district to zero.
Set current district to district in sales record.
DO WHILE current district matches district in sales record.
      And more sales data.
    Accumulate total for a salesman.

```
    Add total for a salesman to total for a district.
ENDDO
Print total for a district.
Skip to a new page.

Accumulate total for a salesman
Set total for a salesman to zero.
Set current salesman to salesman in sales record.
DO WHILE current salesman matches salesman in sales record:
        And current district matches district in sales record.
        And more sales data.
    Add sales data to total for a salesman
    Read sales record
ENDDO
Print total for a salesman.
```

A programmer who is experienced in structured programming should find the specification and expansions just given relatively easy to code. Although some of the loop conditions have only been named (e.g., more sales data), their expansion into code should not pose a great problem. Before doing any coding, however, a little desk checking is often found to be of value.

verification

Seldom can practical programs be completely defined on a single page using the program design language. More likely, the first page of material that is written in that language names the functions that are to be expanded on another page. The first- (or highest-) level page of program design language statements defines the environment of the lower-level function. After the completion of one page in that language, it is often useful to take a checkpont and verify the completeness and correctness of a function that is defined by the program design language. In doing the verification, it may be helpful to list the various combinations of inputs needed to test a routine, in effect, to define—at least in part—what must be done to test the program. In any event, one last thorough examination of a unit of design description before proceeding to lower-level design or coding may save subsequent rework. For example, attempting to process even one record by the example report program reveals the need for a read-sales-record statement before the first DO WHILE at the highest level, i.e., *Summarize sales*.

Experience and conclusions

At MCAUTO, the following major advantages of using the program design language instead of traditional techniques for detailed program design have been observed:

- Ease of writing programs.

- Ease of changing programs.
- Transferability into structured code in a top-down manner.
- Ease of reading programs, especially by nonprogrammers.

The readability aspect contributes to the effectiveness of structured walkthroughs for nonprogrammers. Since the program design language is inherently hierarchical and structured, it also contributes to the success of top-down development and structured programming. Although further experience is needed, it appears that the functional orientation of HIPO also lends itself to expansion into the program design language. Thus the use of the language contributes to the successful use of the other programming techniques.

The systematic application of the program design language is not a cookbook checklist for designing programs. In practice, the individual steps—especially those involved in expanding a design—tend to be done simultaneously, rather than sequentially. Initially, the program designer may be slowed down by his unfamiliarity with manipulating DO WHILEs and IF THEN ELSEs to accomplish his purpose without recourse to GOTOs. With experience, program designs are usually created more readily than otherwise. The resultant designs are typically of better quality than traditional program designs. The better quality of programs designed using the program design language is reflected in relative ease of implementation and maintenance, and by the absence of production errors.

ACKNOWLEDGMENTS

The author extends his thanks and appreciation to the MCAUTO programmers for their interest and perseverence during our mutual learning period. He especially thanks Charles E. Holmes (MCAUTO St. Louis), John E. Hiles (MCAUTO West), and E. Jean Bland (IBM, St. Louis) for the imagination, dedication, and leadership that contributed to the successful adaptation of the methods discussed in this paper.

CITED REFERENCES

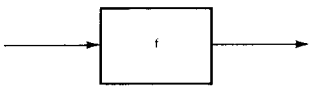
1. F. T. Baker, "Chief programmer team management of production programming," *IBM Systems Journal* **11**, No. 1, 56–73 (1972).
2. C. E. Holmes and L. W. Miller, *Proceedings, 37th Meeting of GUIDE International*, Boston, Massachusetts, October 28–November 2, 1973 (560–575).
3. C. E. Holmes, *Proceedings, 39th Meeting of GUIDE International*, Anaheim, California, November 3–8, 1974 (689–700).
4. *HIPO—A Design Aid and Documentation Technique*, Order No. GC20-1851, IBM Corporation, Data Processing Division, White Plains, New York 10604.
5. *Improved Programming Technologies—An Overview*, IBM Systems Reference Library, Order No. GC20-1850, IBM Corporation, Data Processing Division, White Plains, New York 10604.

6. N. Wirth, *Systematic Programming: An Introduction*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1973).
7. E. W. Dijkstra, "The structure of T.H.E. multiprogramming system," *Communications of the ACM* **11**, No. 5, 341–346 (1968).

Appendix

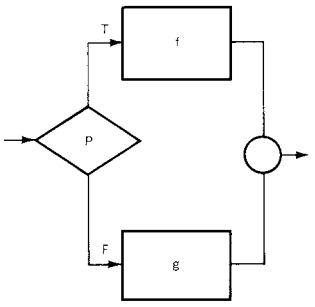
The syntax of the program design language includes provisions for expressing the three basic logic constructs (or figures) of structured programming: SEQUENCE, IF THEN ELSE, and DO WHILE. In the program design language, these constructs have been augmented with the PERFORM UNTIL and CASE constructs. Each logic construct has a definite and simple syntax. In addition to the statement syntax, conventions have been established for the use of indentation and the size of self-contained units of the program design language. The SEQUENCE construct is used to describe any action or work that is followed by the next sequential construct. In control structure forms, SEQUENCE is represented by the function of a subroutine block as shown in Figure 1, where f is the action or work to be done. Syntactically, SEQUENCE represents a simple English sentence, with at least a verb and an object. In practice, the language is most meaningful when action-oriented statements with objects that are natural to the problem are used. Compare, for example, the following sentences: "Print." with "Print XYZ." and with "Print gross sales for salesman."

Figure 1 Control structure for the SEQUENCE logic construct



The IF THEN ELSE construct is used to describe binary decisions. In its most general form, that logic construct is used to describe the conditions under which one of two actions are to be taken. The control structure for IF THEN ELSE is given in Figure 2. The symbol is the predicate (or list of conditions), and f and g are alternative actions. Note that f and g may include any of the logic constructs, and are not limited to being the SEQUENCE construct. The general syntax of the IF THEN ELSE construct is as follows:

Figure 2 Control structure for the IF THEN ELSE logic construct




```
IF      p
THEN    f
ELSE    g
ENDIF
```

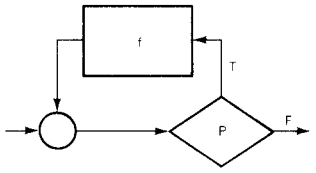
The IF, THEN, ELSE, and ENDIF should always be vertically aligned and displayed in all capitals for ease of reading. When p consists of multiple simple conditions, each condition should be written on a separate line, and all conditions should be vertically aligned, as, for example, in the following way:

```
IF        No more data or
          Different department.
THEN      Print total department sales.
ELSE      Add sale amount to total department sales.
ENDIF
```

The IF and ENDIF conditions are required. When, however, either the THEN or the ELSE clause is not needed, they may be omitted. In other words, the following are syntactically valid forms of the IF THEN ELSE logic construct.

```
IF        p
THEN      f
ENDIF
and
IF        p
ELSE      g
ENDIF
```

Figure 3 Control structure for the DO WHILE logic construct



The DO WHILE logic construct is used to describe the repetition of an action under prescribed conditions (looping). The control structure for DO WHILE is shown in Figure 3, where p is the predicate (or list of conditions) and f is the action to be taken. (Note that Figure 3 is a decision loop in which the action is taken when a condition is true.)

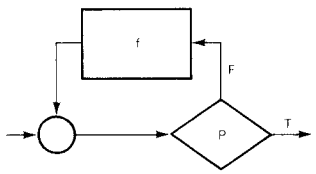
The program design language syntax of the DO WHILE construct is as follows:

```
DO WHILE     p
       f
ENDDO
```

where the DO WHILE and ENDDO conditions are vertically aligned and capitalized. Consider the following pseudo code sequence that is based on the example in the body of this paper:

```
DO WHILE     More data and
                 Same district:
       Accumulate district sales total.
       Read next sales record.
ENDDO
```

Figure 4 Control structure for the PERFORM UNTIL logic construct



The PERFORM UNTIL construct is used to describe looping, when COBOL is the target language for implementation. Control structure for PERFORM UNTIL is shown in Figure 4, where p is the predicate, and f is the action to be taken. PERFORM UNTIL differs from the DO WHILE in that the PERFORM UNTIL loop exits when p is true, rather than when p is false. In effect, DO WHILE p is equivalent to PERFORM UNTIL not p. By using a PERFORM

UNTIL in the program design language, p may be written exactly as it is written in COBOL, thus avoiding the errors that might occur in doing a Boolean inversion of p from the DO WHILE of the program design language to the PERFORM UNTIL of COBOL. The program design language syntax of the PERFORM UNTIL logic construct is given as follows:

```
PERFORM UNTIL     p
        f
ENDLOOP
```

where the PERFORM UNTIL and ENDLOOP are vertically aligned and capitalized. An example fragment taken from the text and expressed in the program design language is as follows:

```
PERFORM UNTIL     No more data or
                  Different district:
        Accumulate district sales total.
        Read next sales record.
ENDLOOP
```

In comparing this fragment with the DO WHILE example, note that the loop conditions have been inverted.

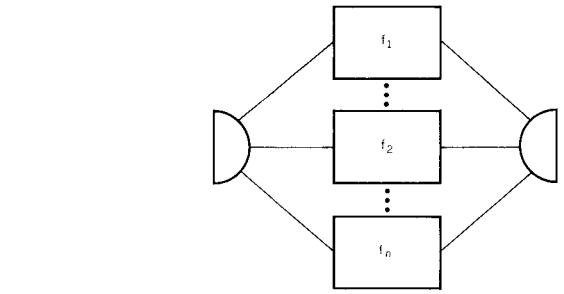
The CASE logic construct is used to simulate a branch table. In the appropriate situation, CASE can be an efficient and effective alternative to multiple levels of nested IF THEN ELSE statements. This construct may be applicable when one of n functions is to be executed, depending on the value of a single variable. The control structure for the CASE construct is shown in Figure 5A. Figure 5B is the IF THEN ELSE logical equivalent of the CASE construct.

The program design language syntax of the CASE construct is given as follows:

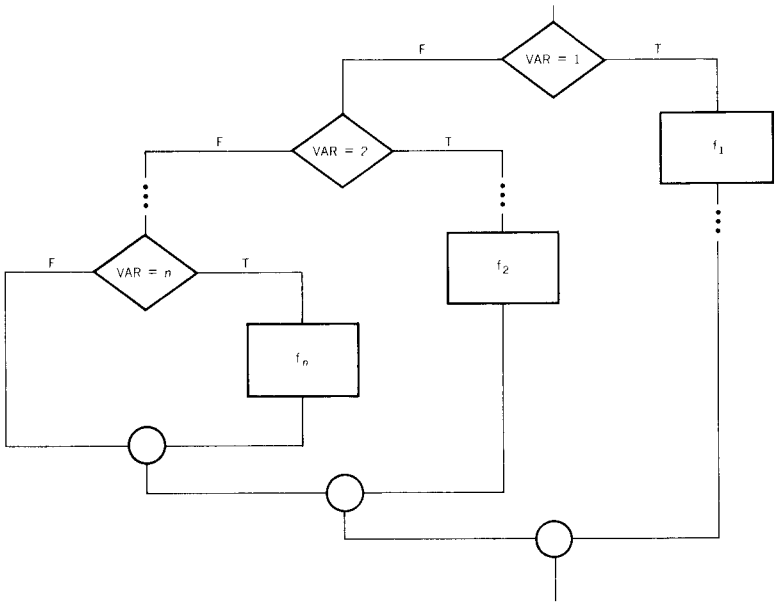
```
CASE     variable  OF
        Value 1:     f1
        Value 2:     Value 3:     f2
        Value 4:     f3
        •
        •
        •
        Value n:     fm
ENDCASE
```

Here, "variable" is the variable to be checked for the various "values," and "value i" is a specific value of the variable to associate with the execution of the function f_i, which appears on the same line. Note that there may be more values n than there

Figure 5 Control structure for the CASE logic construct
A. General f_n case
B. Functional equivalent f_n case



(A)



(B)

are unique functions m, and it is assumed that "variable" has been checked for valid values. Colons are used to delimit the values, as in the following example:

```
CASE        SALES CODE OF
      1:2:  CASH SALE
        3:  REVOLVING CHARGE SALE
        4:  DEFERRED PAYMENT PLAN SALE
        5:  MDSE DAMAGED RETURN
        6:  WRONG MDSE SENT AND RETURNED
ENDCASE
```

The keywords CASE and ENDCASE are vertically aligned and capitalized.

The indentation conventions in the program design language as used in this paper are to align vertically concatenated logic constructs and to indent any nested logic constructs. Two logic constructs are said to be concatenated when one immediately follows the other. The following fragments of program design language are concatenated:

```
IF          p
THEN        f
ELSE        g
ENDIF
DOWHILE     g
      h
ENDDO
```

where DO WHILE follows IF THEN ELSE.

Two logic constructs are said to be nested when one is contained within the other. For example, suppose function f were expanded, then we might have the following nested statements:

```
IF    p
THEN DO WHILE     q
          f2
      ENDDO
ELSE    g
ENDIF
```

where f2 is nested within a DO WHILE, which, in turn, is nested within the IF THEN ELSE. If f2 were expanded into two concatenated sequence constructs, the following structure might result:

```
IF        p
THEN      DO  WHILE    q
              f21
              f22
          ENDDO
ELSE      g
ENDIF
```

In applying the indentation rules, a basic unit of indentation (usually three spaces) should be used consistently. These guidelines, coupled with the use of meaningful and application-oriented names help to make the design easy to read.

The idea of unit size of program design language has been mentioned in the body of the paper. It is based partly on convenience and partly on a perceived, but not well documented observation of human attention span and ability to abstract and

synthesize information. A convention has, therefore, been adopted. Simply stated, the convention is that a single unit should not exceed one page of standard 8 1/2 × 11 inch paper. Furthermore, each logic construct should end on the same page on which it begins. In practice, this results in a package of one-page units where voluminous nested functions are represented by simple names—where they are used—that are then defined in detail on separate pages. Essentially, the program design consists of a number of subroutines that are hierarchically related.

Reprint Order No. G321-5032.