JPL PUBLICATION 84-99

Flowcharting With D-Charts

Donald D. Meyer

January 15, 1985

NASA
National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL PUBLICATION 84-99

Flowcharting With D-Charts

Donald D. Meyer

January 15, 1985

NASA
National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration

ACKNOWLEDGEMENT

The basic constructs for D-Charts were made known to the author as part of a seminar series on Structured Programming conducted by Kim R. Harris at the NASA Ames Research Center during September, 1976.

ABSTRACT

A D-Chart is a style of flowchart using control symbols highly appropriate to modern "structured" programming languages. The "D" comes from Edsger Dijkstra who devised the basic notation. The intent of a D-Chart is to provide a clear and concise one-for-one mapping of control symbols to high-level language constructs for purposes of design and documentation. The notation lends itself to both high-level and code-level algorithmic description. The intent of this paper is to address the various issues that may arise when representing, in D-Chart style, algorithms expressed in the more popular high-level languages. In particular, the peculiarities of mapping control constructs for Ada, Pascal, Fortran 77, C, PL/I, Jovial J73, HAL/S, and Algol are discussed.

CONTENTS

SECTION 1

THE PROBLEM WITH FLOWCHARTS

Flowcharts are held to have two major functions as a means of system or algorithmic description. The first is to assist in the creative process, allowing the designer a graphical means of visualizing program constructs and relationships. This is often expressed in introductory programming texts as: "Coding begins with the drawing of a flowchart." The second function is expository, providing a vehicle for explaining to others the meaning of a high-level design, or of detailed coding. Thus, flowcharts often represent a major portion of the "documentation" of a program.

A standard for flowchart symbols was first proposed in 1963[1] and was revised and extended several times, culminating with the ANSI Standard of 1970[2,3]. It has some utility in describing a data processing function thanks to a large collection of symbols for input/output media and operations. However, the Standard is awkward for expressing certain basic structures which arise in algorithms to be rendered in high-level languages.

The ANSI Standard provides constructs, e.g. diamonds and trapezoids, which map to low-level "machine" processing. As such, they may remain appropriate for describing algorithms written in assembly language. However, for expressing high-level language programs of any significant size, the restriction to ANSI symbols almost invariably produces awkward constructions which obscure the intended meaning. In the process of representing a high-level design with low-level symbology, excessive page space is often required which only further

confuses the representation due to the requirement for off-page connections. Also, the use of closed shapes is often irksome in practice, producing cramped and cryptic prose or statements, particularly in the diamond.

To counter these objections, alternate flowchart mechanisms have been proposed. In particular there is the Nassi-Shneiderman Chart[4], and a variant on the same theme, the Chapin Chart[5]. Both of these techniques define rectangular control structures which are then nested at ever smaller sizes within an overall rectangle representing a complete process. Due to the physical limitations of page size, a designer is forced to produce relatively modest modular constructs "in keeping with good programming practice." In actual practice this imposes a needlessly severe restriction which can lead to a chart of such compactness and complexity that it may pose readability problems, if, in fact, the algorithm represented can be contained at all. Also, this particularly inflexible flowchart style would appear to be of limited value as a design aid. Any modifications will quite likely require that the entire chart be redrawn.

One line of argument contends that a properly indented output listing in a consistent format is adequate for illuminating program flow. This is formalized in a Program Design Language (PDL)[6], in which indentation is used in conjunction with "structured English" to represent a high-level design. However, indentation often fails for logically complicated modules which in the real world can often span several listing pages. This is due to the difficulty of matching a construct identifier, e.g. **do** or **begin**, to its respective end identifier, if the span is long and/or the nesting deep. One would hope for a

representation in which the basic types and interrelationships of the flow constructs present are apparent at a glance. In nearly all cases indentation does not meet this objective.

These problems have led some to abandon flowcharting altogether, perhaps producing them after the fact for "documentation" due to managerial fiat. This process has in fact been automated in programs which can produce prodigious pages of flowchart output of dubious value.

The continuing problem of software visibility still calls for some type of clear, concise graphical representation. D-Charts are intended to meet this need by very pragmatically mapping, one to one, a set of clearly visible symbols to the flow constructs encountered in high-level languages. The basic flow control symbology to be presented is attributed to Dijkstra, having been used by him in some correspondence. The extensions and refinements required to deal with issues that arise in a variety of languages are those of the author.

SECTION 2

THE CASE FOR D-CHARTS

Compared to flowcharts using ANSI Standard symbols, it is claimed that D-Charts are:

- Simpler
- Easier to create, read, and learn
- More powerful and descriptive
- More compact

D-Charts are simpler because they form a notation which reflects the same complexity as high-level programming languages. Traditional flowcharts are more complex since they represent high-level constructs with low-level symbology. D-Charts, with only a few powerful symbols, may be learned quickly, given that they map directly to constructs in "structured languages" such as Pascal and Ada*. In particular, a repetitive loop-control symbol is provided which dispenses with the clumsy, and often misleading, construction forced by ANSI Standard symbols. Finally, D-Charts are more compact, frequently allowing a reduction in the size of the chart, thereby reducing or eliminating lines from one page to the next.

*Ada is a trademark of the U.S. Department of Defense.

SECTION 3

D-CHART SYMBOLOGY

It has been proven that only three basic control constructs are needed to express any computable algorithm[7]. These constructs are sequential, conditional, and repetitive. Each high-level language expresses each of these by a somewhat different syntax, and perhaps provides some additional flow constructs for convenience. The D-Chart flow symbols provide a common representation, regardless of these syntactic differences.

The only "lines" in a D-Chart are those used to show non-sequential control paths, e.g. loops and conditional branches, and for page-to-page flow continuation. In a proper D-Chart (reflecting a properly structured program) no lines go up; all lines either go down or sideways. Any need for rearward control flow can and should be met with loop control symbols. A D-Chart always starts at the top of a page and ends at the bottom, flowing sequentially from page to page if need be, greatly enhancing readability.

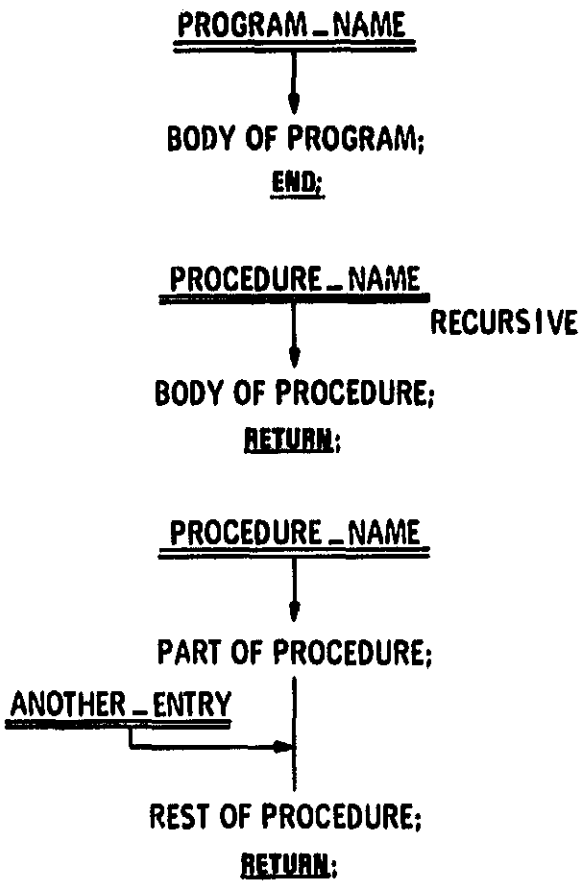
In this paper reserved words (keywords) of particular languages will be indicated in **boldface**. Keywords which can appear in the D-Chart in some particular context, e.g. **call**, **return**, and **cancel**, will be indicated by underlined **boldface**. Also, built-in functions provided by a language, e.g. sqrt, xor, cos, and max, are underlined. Keywords such as **if**, **else**, **for**, **begin**, **do**, **case**, and **continue** are embodied in the flow symbols presented in the following sections, and as such should never appear in a D-Chart except as noted. Each code-level statement should end with the statement terminator of the particular language, typically a semicolon.

3.1 SEQUENTIAL

At the top of the D-Chart the module name is underlined twice, forming an entry symbol. The "module" being represented may be a program, procedure (subroutine), or function; it may even be a general algorithm divorced from any particular programming language. If the module has any special attribute such as reentrant or recursive, this should be included under the module name.

One of the basic tenents of "structured programming" is that the three basic control constructs, as well as complete program modules, have but one entry and one exit. This rule forces clarity and regularity on the program structure, and aids greatly in verifying program correctness. Unhappily two older languages, Fortran and PL/1, provide a mechanism to violate the one input rule for modules by means of an **entry** statement. If for some reason this feature is needed, the entry name is also double underlined, and a line is drawn to that point in the D-Chart where the alternate entry begins.

The final statement of a proper D-Chart should be that of the particular language for the type of module being represented. That is, **end**, **term**, or **close** for programs, and **return** for procedures and functions. For languages in which the return point is implicit at the end of the coding, e.g. Jovial, an explicit **return** should nonetheless be supplied. The following are three complete and proper D-Charts.



In appearance the most significant difference between a D-Chart and an ANSI Standard flowchart is the elimination of closed shapes, e.g. rectangles, diamonds, and circles. Sequential statements are written in free-form, one below the other as follows:

```
statement;
next statement;
next statement;
    .
    .
    .
```

These statements may be at whatever level of abstraction for which the D-Chart is intended. For example:

```
VOL = (PI * R ** 2) * LENGTH;        Exact coding
VOL ← cylindrical volume;            } High level descriptions
Compute cylindrical volume;          }
```

A high-level description may replace one or more code-level statements with one or more declarative sentences or phrases, generally begun with a verb, e.g. form, increment, determine, and set. For a somewhat lower-level description it is suggested that the arrow (←) of APL be used as the assignment operator to generalize the description. In this case the names of actual variables might be mentioned, if appropriate.

For a code-level D-Chart it is recommended that the statements be presented exactly as they appear in the compiler output listing. This will be different from the source in some instances, most particularly HAL/S which outputs an exponent-main-subscript format. For example:

```
VALUE = (M * M1$(*,I)) • V;          Exact HAL/S coding
```

$$\text{VALUE} = (\overset{*}{M} * \overline{M1}_{*,I}) \bullet \overline{V}; \qquad \text{HAL/S listing}$$

Overmarks indicating special types, e.g. vector, matrix, and boolean, should be retained as indicated, if present in the listing. Character strings, bit strings, pointer variables, and non-decimal constants are represented by the syntax of the particular language.

Any labels should be placed adjacent to the statement or flow control symbol to be labeled, preferably to the left. The recommended style is to follow the label with a colon and enclose it with pointed brackets. For example:

```
<search_phase:>  reset search counter;     High-level description
      <128:>  PTR = 1                     Fortran coding
```

In some languages an arbitrary section of coding may be labeled for documentation or as an address reference. In this case the beginning of the block should be labeled as indicated above, and the end of the block should be labeled similarly, but with the colon omitted. In Fortran the statement label (number) may be omitted if it serves only to implement a basic construct, e.g. a "**do** loop."

Some languages permit string replacement for enhanced readability of expressions of choice. For example, this form is called a define-name in Jovial, a symbolic constant in C, and a replace macro in HAL/S. The appearance of a string replacement should be made apparent by dashed underlining. For example, if PRINT is defined as "WRITE(6)", a using statement might be

```
PRINT  X, Y, Z;
```

Any comments desired to annotate the D-Chart should be placed in a convenient location near the section to be described, and denoted either by the syntactic form of a particular language, or by braces {} as a more general form:

```
N1, N2, N3 = FIRST;          /* initialize search pointers */
F = M * (DELTAV/DELTAT);     {get current force}
```

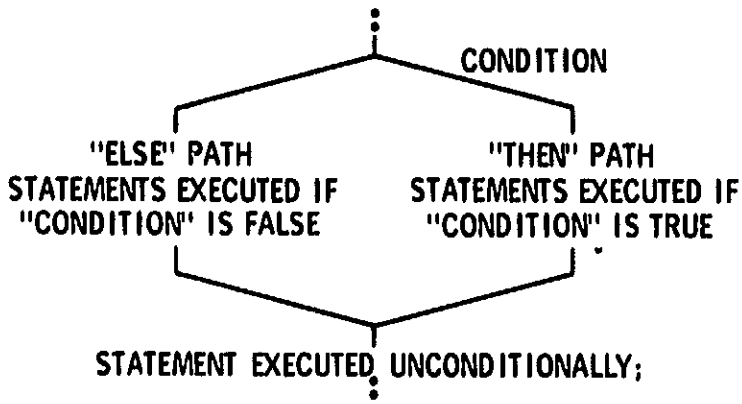
A high-level D-Chart will consist primarily, if not entirely, of "structured comments" which replace code segments. As such, these comments do not require the special denotation of code-level D-Charts.

3.2 CONDITIONAL

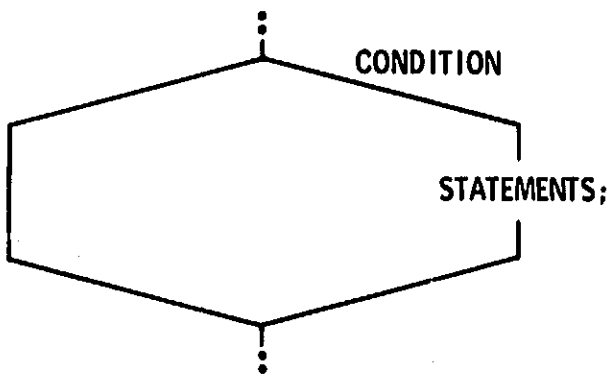
Two D-Chart symbols are provided for conditional branching. The first covers the bi-directional decision, **if-then-else**, while the more general form is handled by the **case** construct.

3.2.1 If-Then-Else Symbol

The following symbol is used:

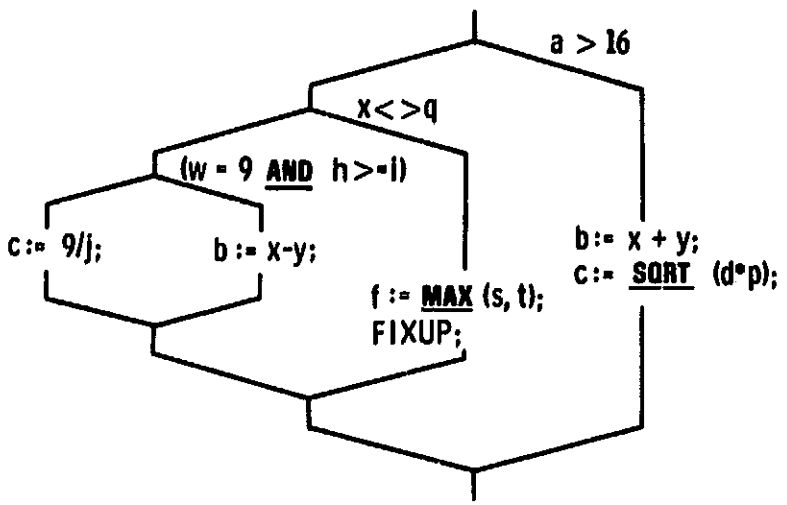


The "**else** path" may be null, in which case a line is drawn vertically joining the top and bottom portions of the symbol.



Either, or both, of the **then** (true) or **else** (false) paths may be any control structure which obeys the general structured rule of "one input path, one output path." Each **if-then-else** symbol should have an explicit join which causes the input and output paths to be aligned on the page. This join is made explicit in Fortran and Ada by **end if**, and replaces those keywords. Note also that this symbology removes any ambiguity arising from a "dangling **else**."

The following is a Pascal example of a nested set of **if-then-else** symbols which represents an "**else if** chain" (**elsif** in Ada, **elif** in Algol 68), with an **else** terminator:



This example also illustrates that it may be desirable to enclose the "condition" in parentheses in order to avoid confusion between a statement to be executed and a condition to be tested, should statements and conditions be in close proximity.

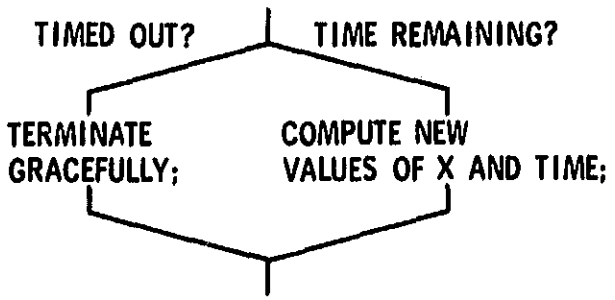
Algol, C, Pascal, and PL/I provide, in some form, a conditional assignment statement. These statements may be long and complex as in the following example from C in which the "condition" is to the left of the "?" and the

"true" value for z is to the left of the colon and the "false" value to the right:

```
z = ((q >= b) && (f != 5)) ? 19/(i+j) : 7 * x;
```

For the sake of generality and clarity, it is recommended that statements of this sort should also be represented by the **if-then-else** symbol and not as assignment statements.

Since the **if-then-else** is a two-way branch, the condition written on either side of the symbol implies that the other side, usually blank, is taken when the logical inverse of the "condition" is true. Both sides may be labeled if desired for clarity or documentation. For example:



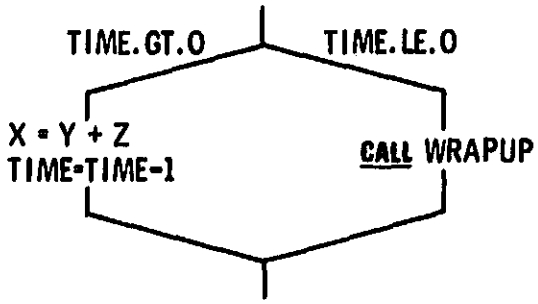
This example also illustrates the conditional form to be followed in a high-level D-Chart. That is, the condition should be posed as a question followed by a question mark. Also, while a code-level D-Chart should follow the syntactic forms imposed by a particular language, a high-level D-Chart may use more meaningful conditional and logical symbology, e.g. ">" instead of ".GT.".

It is not necessary to strictly mimic the coding in a code-level D-Chart as to which is to be considered the "true" path. That is, if the chart

arrangement works better by reversing the logical sense, then do so. This particularly becomes an issue for the escape symbol to be discussed later. If the logical sense is reversed, both the "**then**" and "**else**' conditions should be indicated. Note that in this case the "**then** path" might be null. For the high-level example just given, the Fortran code might have been:

```
IF (TIME.GT.0) THEN
   X = Y + Z
   TIME = TIME -1
ELSE
   CALL WRAPUP
END IF
```

In this case the "reversed **if-then-else**" would be:



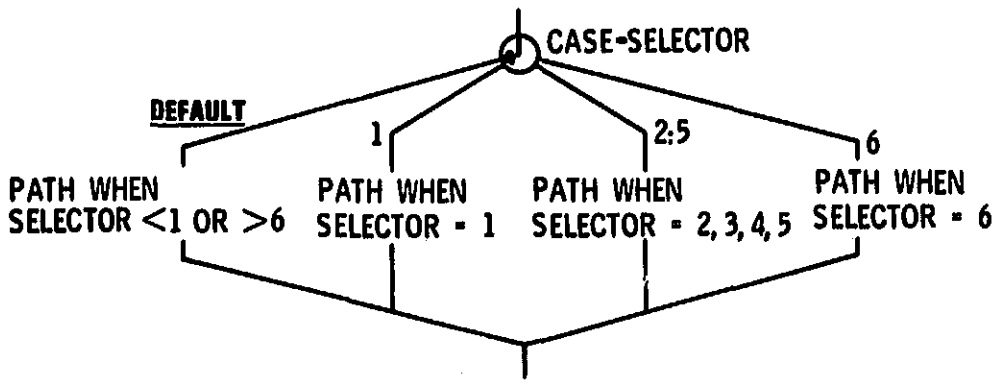
3.2.2 Case Symbol

The **case** symbol provides a multiple path branch such as Ada`s **case** statement, C`s **switch**, and Fortran`s Computed **go to** or Arithmetic **if**. All of the languages considered have a mechanism to provide this form, albeit clumsily for some, e.g. in PL/I (**go to** subscripted-label-variable). The "one input path, one output path" requirement is met by requiring that all branches of the **case** return to a single point following the **case** symbol.

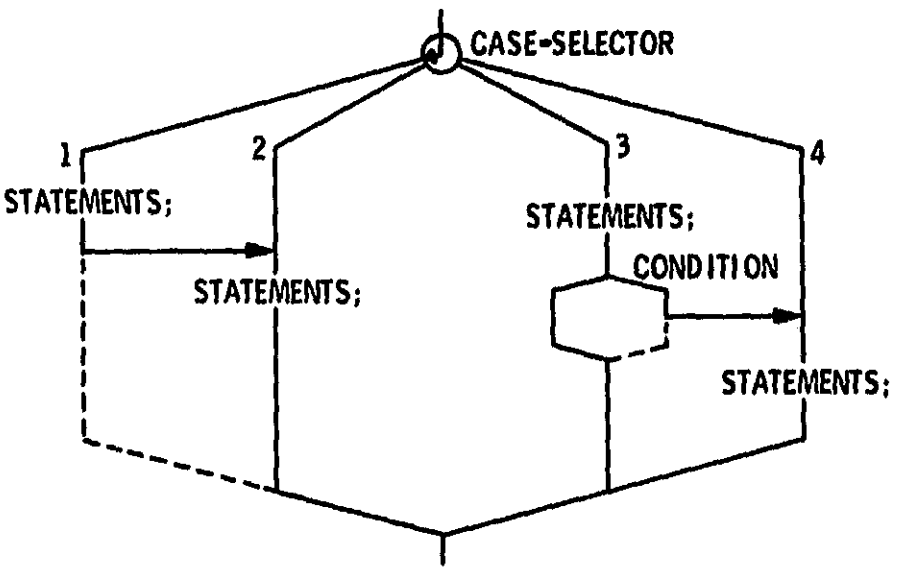
The "case-selector" which determines the path of the n-way branch will, in general, be an expression which evaluates to an integral value. Successive values then match to a statement or statement block that follows. For most languages the case path is specifically labeled by the "integral value," or values, while others allow for a range of values, e.g. Jovial's bound pairs. Also, Ada, Jovial, and Pascal provide that the case selector can be an enumerated ordinal type, e.g. COLOR, which has members, e.g. BLUE, RED, and YELLOW. Due to these possibilities each branch line of the **case** symbol should be explicitly labeled as to its corresponding selector value or case label list.

Also, some languages provide a case path for an out-of-range selector value. It should be labeled as such, and is specified by **default** in C and Jovial, **else** in HAL/S, and **others** in Ada. In the absence of an out-of-range identifier, the action to be taken for an unspecified path will be assumed to be that of the language in question, perhaps a branch to the statement following the **case** symbol terminator..

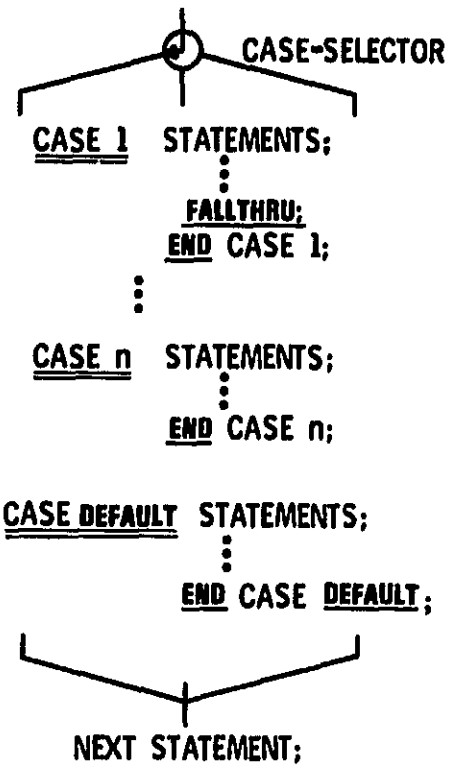
The **case** symbol uses a rotary switch notation. The following example illustrates the form to be followed if the number and complexity of the case paths are small.



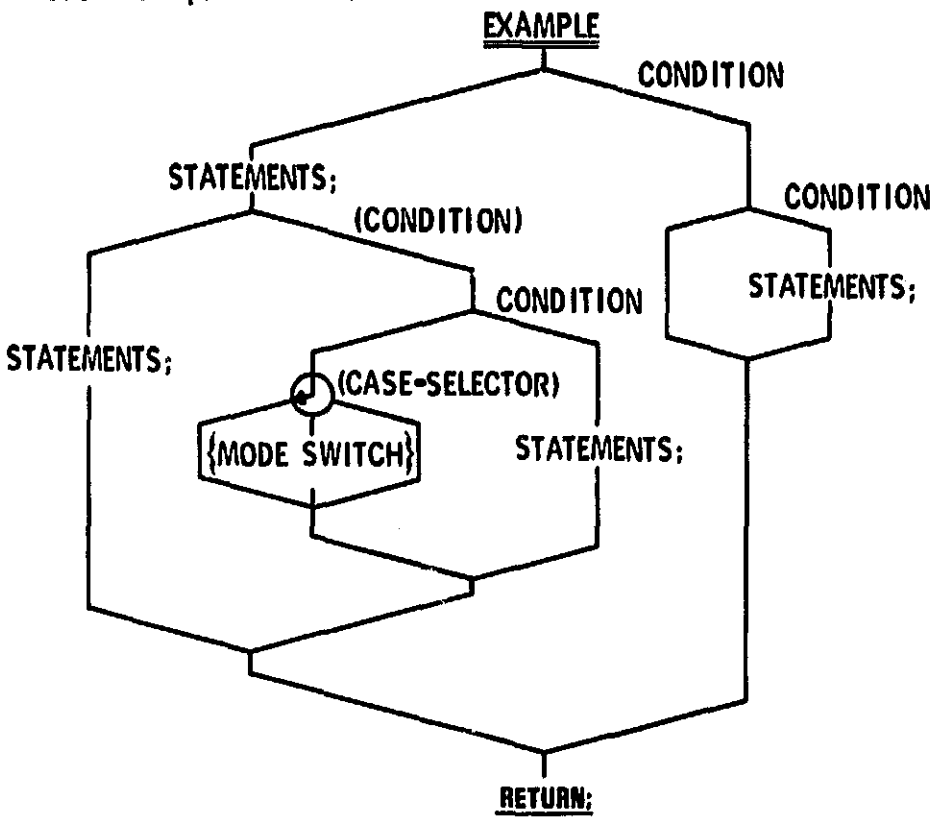
An additional complication arises for those languages which allow execution to proceed to the next case path following the one specified by the case-selector. This action is specified in Jovial by the **fallthru** statement, in C by the absence of a **break**, and in Fortran and PL/I by the absence of a (**go to** end-of-all-cases) at the end of a case path. This may be a continuing process depending on the presence or absence of **fallthru**, **break**, or **go to**. In this situation the case paths are arranged so that the statements in a path to be "fallen to" are lower than those in a path to be "fallen from." A side arrow is drawn to indicate the flow. These paths (to and from) will necessarily be adjacent. Dashed lines should be supplied to indicate the full form of the **case** symbol, with the understanding that a dashed line is an impossible path. A "fall through" may also arise from a conditional construct.



If the number and complexity of statements and control constructs in each case path are manageable for a one column per path format, then the above forms should be used. However, if the number of paths is large, or the statements and logic in each path are complicated, then the columnwise approach is impractical. In this situation a better approach is to place each case path one after another below the **case** symbol. Each path is labeled by underlining the selector value or values which will cause its execution. In this style any **fallthru** or **break** statements will have to be indicated explicitly and underlined. However, for Fortran and PL/I the **go to** at the end of a case path would not be given, as a branch to the end of the **case** is implicit in the **case** symbol, unless otherwise indicated. The **case** symbol, top and bottom, is formed separately with a 3-way symbol regardless of the actual number of paths.



If the number of cases is very large, and/or the logic in the paths is extensive, it may be necessary to consider each case path as an "internal procedure" which stands alone following the entire D-Chart. This is particularly true if the **case** construct is embedded in a complicated logical nest. Each **case** symbol should be clearly commented and cross-referenced to its set of "internal procedures."



Case 1 of Mode Switch

statements;

end Case 1, Mode Switch;

•
•
•

Case n of Mode Switch

statements;

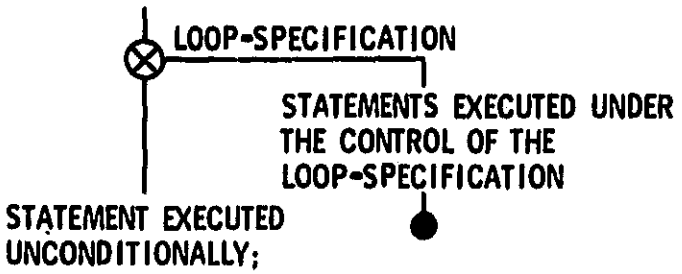
end Case n, Mode Switch;

As with the **if-then-else**, the elements of a **case** construct may be a high-level free-form description. For example, the case-selector could be a description of the selection criteria, and each case path could have the form of declarative sentences describing the entire path. Or, prose could be used in concert with D-Chart symbols to provide a high-level exposition of the path algorithm.

3.3 REPETITIVE

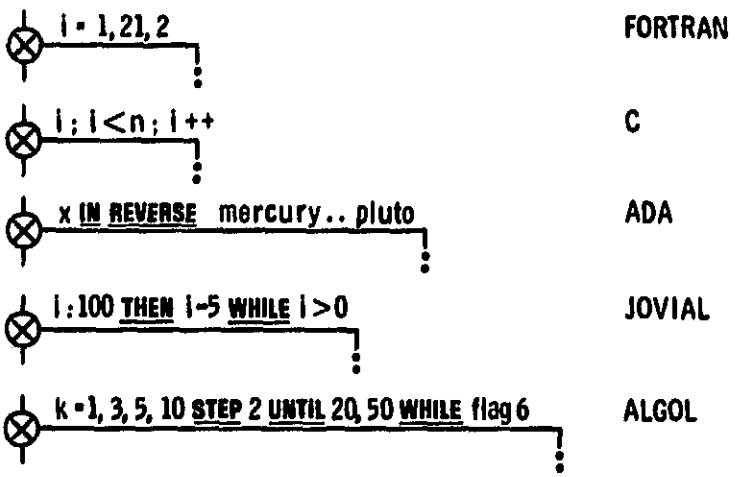
The repetitive, or iterative, loop is the third of the basic constructs required for devising computable algorithms. Loops come in two basic categories, enumerated and conditional. The discrete and incremented-by-step types constitute the enumerated class, and **while** and **until** the conditional. Some languages permit hybrid combinations of these four. All of the languages discussed provide at least an incremented loop.

D-Charts use a clearly visible switch-dot notation to represent a loop. The clumsy decision-diamond/back-arrow form found in ANSI Standard flowcharts is eliminated. The general form is as follows:



To be explicit, the meaning of this construct is as follows. When the switch (⊗) is encountered, the flow comes under the control of the loop-specification for one of the four loop types or a hybrid. The dot (●) is the loop-end symbol and, when encountered, execution returns to the switch for re-evaluation of the loop-specification until satisfied. Once satisfied, control passes to the next statement following the switch. For notational consistency the loop-specification sidebar should always extend to the right. Keywords which terminate a loop, e.g. Ada's **end loop**, are replaced by the "dot."

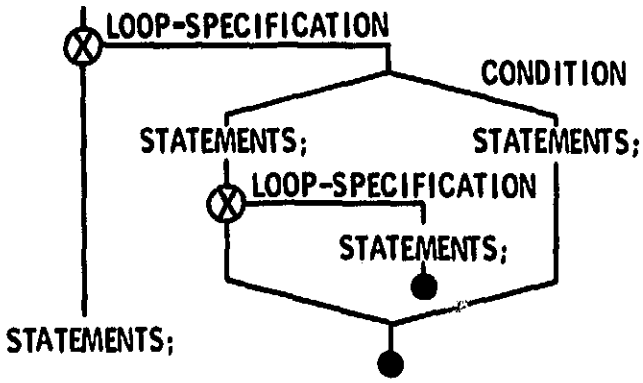
The languages considered use a wide variety of syntactic forms and keywords to perform the same task. For example, an enumerated loop is typically indicated by **do or for**, while Ada uses **for...loop.** In all cases these differences are hidden by the loop symbol, being replaced by the same switch-dot notation. For a code-level D-Chart the loop-specifications should follow the same syntactic form as that of the languages rendered. Some examples follow:



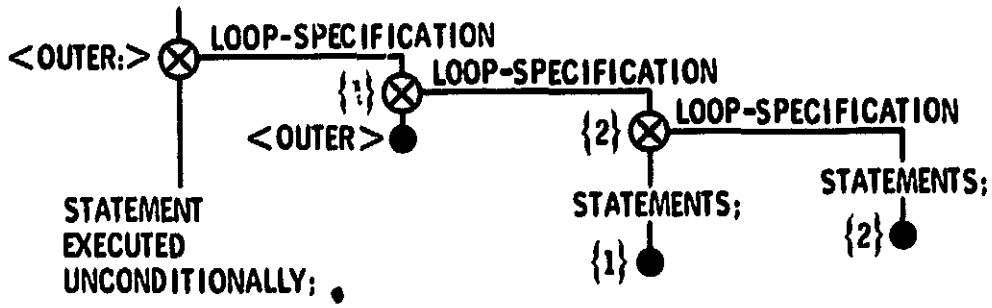
The last two examples illustrate hybrid loop-specifications. In fact, the Algol example uses all four loop forms! A basic Ada **loop** which requires some loop-body construct for termination might be represented as:

In a high-level D-Chart, the loop-specification would take a prose form. For example:

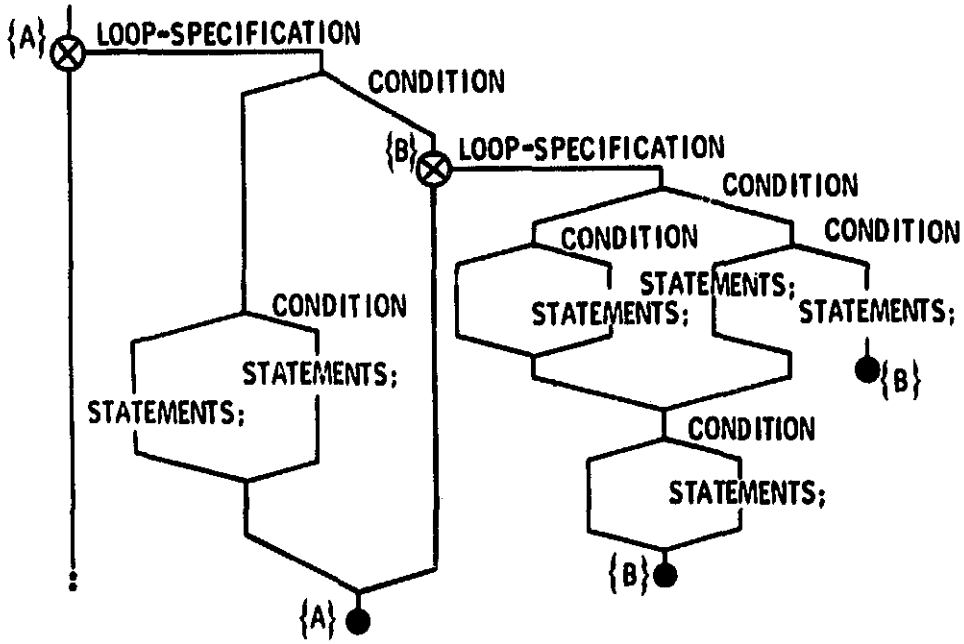
The statements within the loop may be embodied in any of the control structures which follow the general "one input path, one output path" rule. For example:



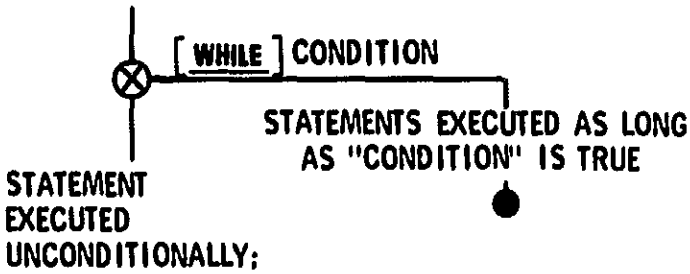
When the D-Chart involves nested loops in close proximity, some form of labeling should be supplied to clearly relate the switch-dot pairs. If the loop is labeled in the coding, the labeling convention previously discussed should be used. Otherwise, the labeling should be done by comments, where the "comment" need not be more than a single letter or number. Also as a matter of style, the dot should be placed as near to its switch as possible.



The HAL/S and C languages provide for more than one loop end, that is, more than one dot per switch. In HAL/S this is accomplished by a **repeat** statement, and in C by **continue**. The action at each dot is as before: revert to the switch for re-evaluation of the loop-specification. In this circumstance, labeling should always be supplied to relate all loop ends to their switch.

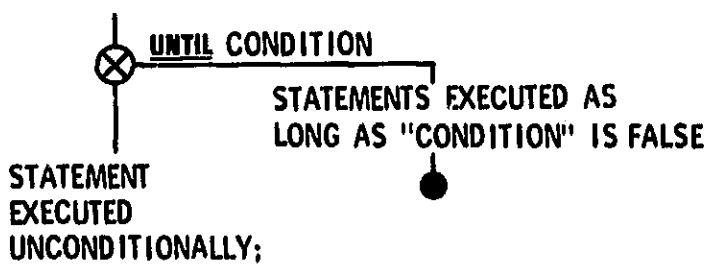


The conditional loop class requires further discussion. In the case of a loop-specification consisting only of a **while** condition, it is necessary to state only the logical expression, preceded optionally by the keyword **while**.



The implication is that the "condition" is tested at the top of the loop as the condition to continue loop execution. This is the case for all the languages considered which have a **while** form. However, C has an additional **while** form in which the test for loop continuation comes at the dot, or dots, not the switch. In this case the prefix **do while** is mandatory.

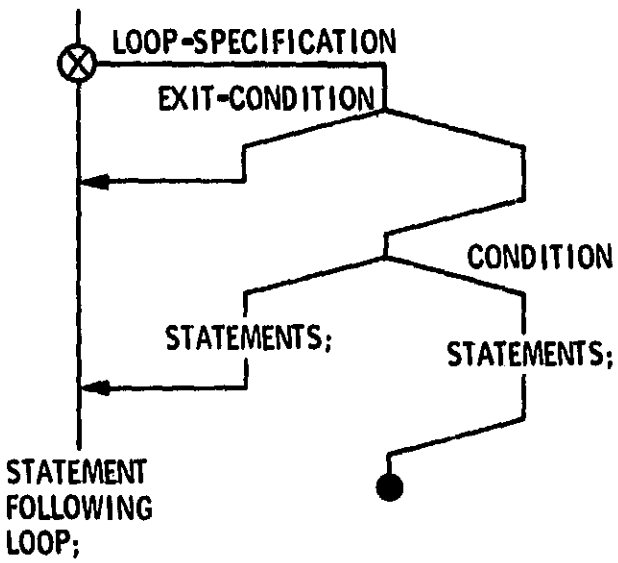
The **while** form accounts for the great majority of exclusively conditional loops. However, there is in addition an **until** form provided by several languages. In this case the test is made at the end of the loop as the condition for loop execution to cease. The conditional expression is in fact placed at the loop end in Pascal. For notational consistency, all loops of this type should be represented by the same form. That is, the **until** keyword (**repeat until** for Pascal) must be provided, with the implication that the test actually occurs at the "dot."



3.4 IRREGULAR FLOW

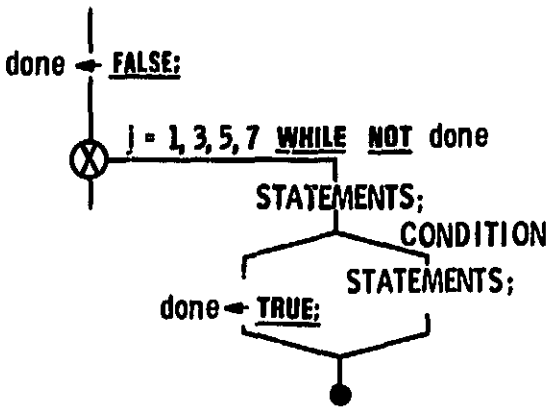
In addition to the three basic control structures, each language provides the means for constructing other flow paths, if nothing else, through use of the infamous **go to**. Jovial, Ada, and HAL/S provide an **exit** statement for loop escape, while C has **break** for transferring out of any of the control structures previously discussed.

D-Charts provide an escape symbol which indicates the termination of a loop at a point other than the evaluation of the loop-specification. The escape symbol has the form of an open **if-then-else** symbol with a side arrow. In the following example the upper escape symbol illustrates a conditional **exit** statement, such as Ada's **exit when**. The lower escape symbol illustrates an **exit or break** terminating a sequential series of statements:



In a proper D-Chart an unlabeled escape may lead only to the statement following the loop in which it is contained, thereby satisfying the "one in, one out" rule. The escape construct can be avoided at the cost of a spurious

boolean variable and two assignment statements in languages which permit hybrid loop-specifications. In this case, the escape path is directed to the loop end as follows:

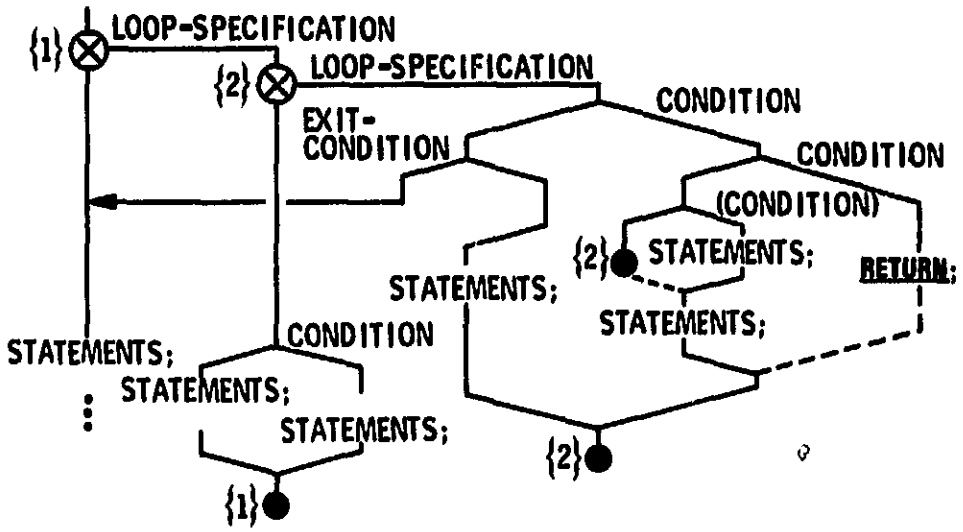


Some languages, e.g. Ada and HAL/S, permit a labeled escape which causes the **exit** of a nest of constructs to a forward point. HAL/S generalizes this to allow an **exit** from any labeled code block bracketed by **do** and **end**, thereby producing a forward **go to** by another name.

Then too, each of the languages provides the traditional (**go to** label), for which some may find a need for reasons of efficiency or perversity. Arguments have been made against its need[7] and desirability[8].

When a **go to**, or an alias, is to be represented, it should be by a directed line to the next statement to be executed. The use of a rearward **go to** is extremely bad form, and should be avoided at all cost. Every attempt should be made to avoid crossing lines. For example, the sense of an **if** test should be reversed if that prevents the escape path from crossing other lines. If the D-Chart is intended to reflect exactly the as-is coding, this reversal should be made clear by indicating the condition and its logical inverse on both sides

of the **if-then-else** symbol. The recommended style for unavoidable crossing lines is as follows:



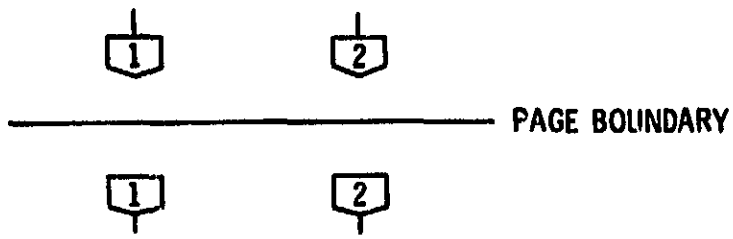
Despite the advisability of a single **return** to meet the "one in, one out" rule, it is realized that **returns** have a way of appearing elsewhere as well, thereby producing a flow break. Also, the loop **repeat** or **continue** discussed earlier will cause a flow break. As illustrated by the above example, it is often desirable to complete an **if-then-else** symbol with dashed lines to remove any possible confusion as to the intended construct. The dashed line implies that the path is logically present, but impossible to traverse.

3.5 LINE CONTINUATION

In order to encapsulate processes at a level of "intellectual manageability," attempts have been made to mandate small modules, perhaps limited to two listing sheets. Nonetheless, it is realized that in the real world the D-Chart for a module will often span several pages. This, of course, requires some mechanism for line continuation to the following page. Line continuation

should never be permitted laterally, that is, it should occur only at the tops and bottoms of pages and never at the sides. This forces a coherent, one-way flow for the D-Chart. In the case of a construction with parallel nesting to extreme depths where this may be a problem, consideration should be given to replacing the logic of some paths with calls to fictitious "internal procedures." These procedures should be clearly labeled as such, have a double underlined invented name as for any procedure, and placed at the end of the entire D-Chart.

The symbol for line continuation has the form of a "homeplate" (⬠). A letter or number should be placed in each symbol to make explicit the line continuation. The number/letter pair across the page boundary should be unique and, if at all possible, the lines continued should be physically aligned on the two sheets. If a given line spans an entire sheet without containing any statements or constructs, then the same number/letter should be carried forward until something occurs in that line. On both the "from" and "to" sheets the continuation symbol should point downwards in the direction of the logical flow.

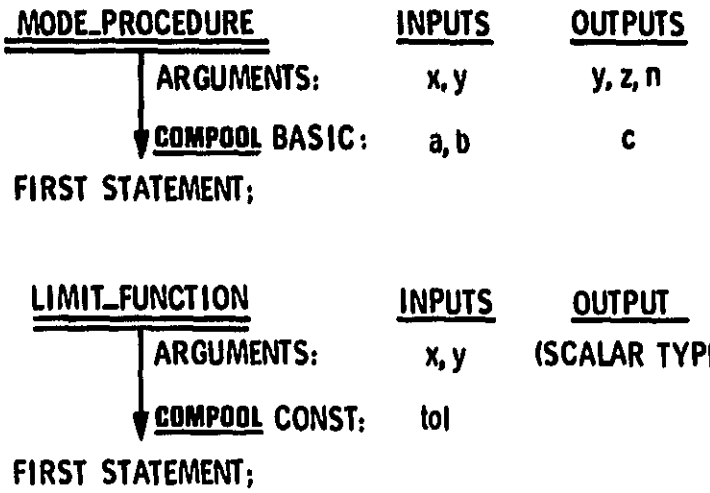


SECTION 4

DATA AND PROCEDURAL LINKAGES

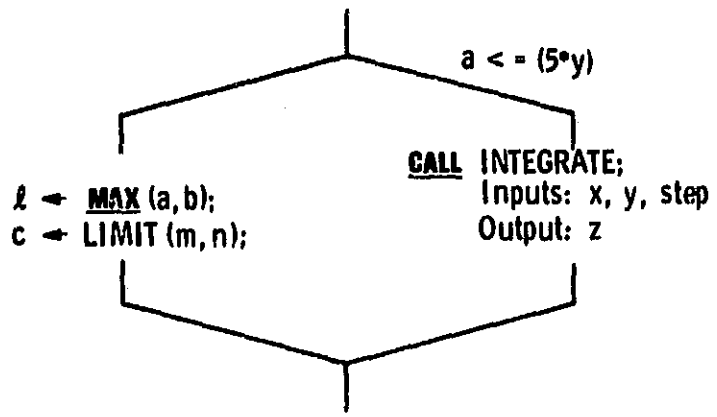
This final section deals with the techniques of handling the interface between the module being D-Charted and the rest of the operating environment. It should be recalled that any flowchart represents executable statements and the flow constructs in which they exist. The side issue of data definition for the module can be handled in several ways. The simplest is not to address the matter at all in the D-Chart, on the basis that the data will be defined by some other mechanism, e.g. a set of HIPO (Hierarchical Input Process Output) diagrams, data flow diagrams, and/or a data dictionary. This must include the definition for all "ordinary" variables, as well as enumerated types, replace strings, the templates of structured types, and package names in the case of Ada. All of these would be prefaced to a self-contained D-Chart of a program module, perhaps categorized according to type, along with other pertinent attributes as desired.

A D-Chart for a procedure or function should, at a minimum, have at its top a listing of all variables which are input and output by argument passing. The data type returned by a function should also be specified as an output. Those variables which are input and output through a **compool**, **common** or **external** mechanism should be separately indicated as well. Variables not so listed could then be assumed to be internal to the module.



Invocation of a function should be presented as it appears in the coding since the inputs (values passed to the function) and single output (value returned from the function) are explicit. For a procedure (subroutine), the various languages provide a number of syntactic forms for distinguishing between input and output arguments, or perhaps make no distinction at all. A procedure call can be indicated as it appears in the coding, or, for more generality, with the input and output arguments clearly listed.

For example:



Note that a "library" function name is underlined, while user-defined function and procedure names are not. Of the languages considered, all but Fortran, PL/I, and HAL/S invoke both functions and procedures only by stating their names. However, a **call** may be prefaced to the procedure invocation for these as well, for the sake of clarity and to generalize the description.

An additional problem arises in that all the languages discussed, except Fortran, permit the declaration of variables in a limited scope smaller than a complete module. For example, in HAL/S variables can be declared **temporary** and exist only in the enclosing block of their definition. If this is an important issue due to duplicate names in nested blocks, then the scope of such variables will have to be indicated by comments adjacent to these blocks.

Of the languages discussed, only Fortran, HAL/S, and PL/I provide input/output statements as intrinsic operations. The others typically provide built-in functions for the purpose. In either case, the operation keyword or function name is underlined as always, e.g. **writeln** and **read**. Any ancillary nonexecutable statements, e.g. **format**, **file**, and **namelist**, may be included, if desired, at an appropriate location in the D-Chart.

HAL/S includes a repertoire of statements for real-time control, e.g. **schedule**, **cancel**, and **signal**. Syntactically these take the form of pseudo-function invocations in which a system action is performed, rather than a value returned. They should appear in a D-Chart as they would in the coding.

Ada, PL/I, and HAL/S provide for user-defined error recovery through **exception, on, and on error** statements, respectively. Also, Ada and PL/I have

extensive facilities to deal with concurrent processing (tasking), e.g. **select**, **accept**, and **post**. The flowcharting mechanisms that would be needed to deal with the intricacies of these two topics is beyond the scope of this discussion.

SECTION 5

CONCLUSION

A symbolic notation has been presented for the graphical representation of the basic flow constructs of a number of popular high-level languages. It has been the author's experience in a number of organizations that programmers faced with the daily task of writing and reading programs take very naturally to the D-Chart style of representing algorithms. In one instance their usage prevailed by popular demand even in the face of a managerial dictate to stay with "standard" forms. It is hoped that the D-Chart style of structured flowcharting will be found useful in alleviating the problem of software visibility.

SECTION 6

REFERENCES

1. R. J. Rossheim, "Report on Proposed American Standard Flowchart Symbols for Information Processing," Comm. ACM, Vol. 6, No. 10, Oct. 1963, pp. 599-604.

2. "ANSI, Flowchart Symbols and Their Usage in Information Processing, X3.5-1970," American National Standards Institute, New York, 1971, 17 pp.

3. N. Chapin, "Flowcharting with the ANSI Standard: A Tutorial," Computing Surveys, Vol. 2, No. 2, June 1970, pp. 119-146.

4. I. Nassi and B. Shneiderman, "Flowcharting Techniques for Structured Programming," SIGPLAN Notices (ACM), Vol. 8, No. 8, Aug. 1973, pp. 12-26.

5. N. Chapin "New Format for Flowcharts," Software Practice and Experience, Vol. 4, No. 4, Oct.-Dec. 1974, pp. 341-357.

6. S. H. Caine and E. K. Gordon, "PDL - A Tool for Software Design," AFIPS Conf. Proc., 1975, pp. 271-276.

7. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules," Comm. ACM, Vol. 9, No. 5, May 1966, pp. 366-371.

8. E. W. Dijkstra, "GO TO Statement Considered Harmful," Comm. ACM, Vol. 11, No. 3, Mar. 1968, pp. 147-148.