SLAC-PUB-1530
January 1975

STRUCTURED CONTROL IN PROGRAMMING LANGUAGES*

Charles T. Zahn, Jr.

Computation Research Group
Stanford Linear Accelerator Center
Stanford, California 94305

*This work supported by the U.S. Atomic Energy Commission under contract AT(043)515.

Prepared for the 1975 National Computer Conference, May 19-22, 1975.

STRUCTURED CONTROL IN PROGRAMMING LANGUAGES

Conceptual Distance

Solving a problem with the aid of a computer involves the construction and execution of a program described by a linear piece of text. First, the problem-solver (programmer) translates his problem into a precedural solution embodied in a static program text, written in a programming language. Then a computer is caused to perform a dynamic sequence of actions in accordance with the commands in the program text. The reliability of this two-stage problem solution (i.e., the likelihood that the actions performed really provide a solution of the problem) depends on the degree to which the program text mirrors the possible action sequences that it causes, as well as the problem solution that it purports to implement. It is useful to speak of the "conceptual distance" between program text and action sequences or between problem definition and program text. The programmer who wants some measure of confidence in the reliability of his program must bridge both these conceptual distances. It follows that a major goal of programming language design should be to help reduce both these distances.

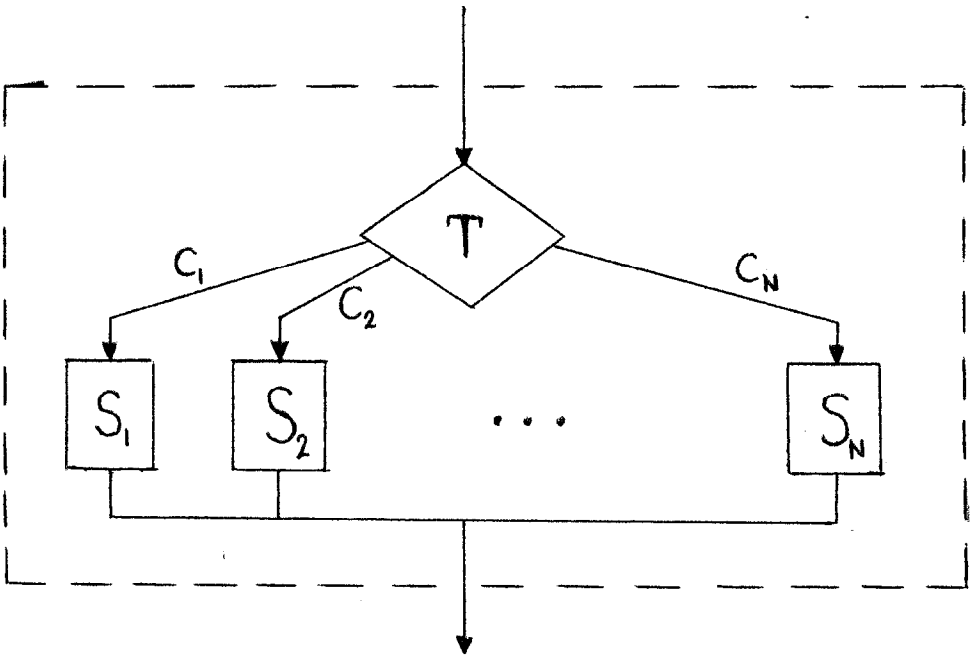
Structured Control

Structured programming[1] is a systematic step-wise method of program composition which can be used to conquer the distance between problem and program by chopping it into bite-sized pieces and employing abstraction as a mental aid to control the problem of complexity. It reduces the distance between program text and action sequence by employing in the program text only those forms of sequence control which allow an easy visualization of the possible action sequences from an inspection of the static linear program text. The control structure includes sequential grouping of commands as well as command selection

(if statements and case statements) and repetition (while and repeat statements). Enumerative reasoning and mathematical induction are available mental aids for understanding the action sequences evoked by programs restricted to these forms of sequence control. These considerations suggest a control structure limited to sequential grouping, selection and repetition.

Problem-oriented Control

Unfortunately, the story doesn't end there because, in spite of the immense advantages of the restricted control, it is still not adequately problem-oriented. This is true even when the control structure is extended by a simple for statement and recursive procedures and functions. One of the commonest situations in programming is the need to select one of a finite set of commands, using some selection mechanism, each of whose outcomes corresponds to a unique command from the set. The following general flowchart models the control:



In this flowchart, $C_1, C_2, \ldots, C_N$ are constants of some finite type, each S_k is a command (statement), and T is a "test" or inspection of program variables whose execution terminates by selection of one of the C_k as its outcome.

The special case (N=2, C_1=*true*, C_2=*false* and T evaluates a logical expression B) represents the familiar control form *if* B *then* S_1 *else* S_2. The case (C_k=k for $1 \leq k \leq N$ and T selects that constant C_k equal to the value of an integer expression E) represents Hoare's integer case statement with the syntax *case* E *of* $(S_1;S_2;\ldots;S_N)$. This expression-driven case statement has been generalized[2] and implemented in PASCAL.[3] By allowing constants C_k and expression E to correspond to any finite type (especially programmer defined types like Color whose 4 constant values might be Green, Blue, Red, Black), the conceptual distance between problem and program can be greatly reduced.

There remain situations in which the selection is not conveniently reduced to an expression evaluation, and T must be a compound command which returns a value C_k. It also naturally occurs that at certain places within T, it becomes clear which value should be selected and an immediate termination of T is entirely appropriate. Recent versions[4] of the programming language BLISS[5] extend the restricted control by allowing any compound statement to be labeled; then a statement of the form *leave* L *with* E, causes immediate termination of the enclosing statement labeled L and returns E as its value. It is, therefore, easy to implement the more general selection mechanism T within BLISS. A more recent proposal[6,7] for extending the control is an event-driven case statement of the form

until C_1 *or* C_2 ... *or* C_N *do* T *then case* $(C_1:S_1;\ldots;C_N:S_N)$

with event statements C_k within T, causing immediate termination of T and selection of C_k. Each C_k is an identifier or name created by the programmer to provide a problem-oriented description of what the program is doing. The syntax for this generalized case statement was motivated by considerations of writing and reading programs in a top-down fashion. The number of similar proposals for a termination mechanism (see the survey by Knuth[7]) shows the uni-

versal need for such a programming device. Other common situations requiring an explicit termination mechanism are repetitions of a command sequence where the detection of the termination condition naturally occurs midway through the sequence and the handling of error conditions which have various degrees of severity.

Repetitions with a Control Value

It is a common need in programming to repeat a given compound command once for each of a well-defined finite sequence of values, where that value is accessable (but not changeable) within the repeated command. When the programmer's intent is exactly reflected in this special form of repetition, there is a great gain in clarity when the program text employs a special syntax to indicate the repetitive pattern. Certainly, there should be a repetition like

repeat for V from E_1 {upthru/downthru}E_2 do S(V)

where V is a variable of ordered finite type and E_1,E_2 are expressions of that type. This is the form (with slight differences in syntax) of for statement implemented in PASCAL.[3,8]

Serious consideration should be given to extensions of the for statement to cater for progressions of values defined by more general successor functions. For example, the programmer who builds sequences using records and references is helped immensely by statements like

repeat for R from Start by Next upto null do S(R)

where R is a reference variable whose values are Start, Next (Start), Next (Next(Start)), etc., up to but not including null. The use of words upthru, downthru, upto is an attempt to reduce the ambiguity that results from not making explicit the distinction between inclusion or exclusion of the final item.

Procedural Mechanisms

Procedures and functions, with carefully designed parameter mechanisms, are now more widely appreciated as beneficial tools for program decomposition and the embodiment of problem-oriented abstractions. They are thus helpful to the programmer in his task of bridging the conceptual distance between problem and program; that is, when their use is not discouraged by considerations of efficiency. The programmer should be allowed to attach the macro option to any procedure or function invocation, and thereby feel free to use them as purely structuring tools without the run-time overhead often implied by the closed subroutine.

The main difficulty in the use of procedures and functions is that the conceptual distance between program text and dynamic actions is often increased by mysterious parameter mechanisms and side-effects.[9] The axiomatic definition of procedures and functions in PASCAL[8] can be interpreted as a suggestion that procedure parameters be classified as constant or as update, while function parameters are restricted to constant. A constant parameter represents a constant value determined by an actual parameter expression at the time the procedure or function is invoked. This value may not be altered by the procedure or function. This has usually been referred to as "call by value". An update parameter represents a program variable whose value can be altered or inspected by the procedure. The actual variable being inspected and altered is the one whose name is given as the actual parameter in the procedure invocation. It would probably be an aid to program clarity to distinguish a third class of result parameters which may not be inspected (since they are presumably as yet undefined!), but which are expected to be assigned values by the procedure. Of course, result parameters would not be allowed for functions.

The program text of a procedure or function should indicate all those global (i.e., non-local, non-parameter) variables which are referenced within it with a textually clear distinction of those which are potentially alterable by the procedure. No functions should alter any globals. Whether this additional program documentation is made the responsibility of the programmer or a helpful compiler -- in either case it provides crucial textual evidence to aid the programmer in visualizing the possible dynamic actions caused by a given invocation of the procedure or function. Another important restriction[8] is the disjointness of the set of alterable parameters and global variables. Failure to comply with this restriction may cause very nasty and subtle errors.

It has recently been proposed by Hardgrave[10] that a keyword, rather than positional notation for the correspondence between formal and actual parameters, would have several nice advantages, one of which is the obvious textual clarity of the programmer's intent. In the case of procedures and functions with long parameter lists, there is a disturbing potential for erroneous parameter communication even in a highly typed language. By allowing default actual parameters[10] for certain formal parameters to be explicitly given within the procedure declaration, the textual length of the invocation can often be kept reasonably small in spite of the apparent verbosity of the keyword notation. It is also possible to add a new parameter without altering previously written invocations of the procedure -- a potentially non-trivial advantage in a large software project requiring modifications through time.

Recursive procedures and functions should be allowed since they reflect problem solutions whose reprogramming without recursion involves considerable conceptual distortion and, therefore, increases the conceptual distance between problem and program. In a similar way, there are certain problems which are most naturally solved by two or more procedures whose relationships to one

another are more symmetric than the normal hierarchical procedure relationships.[11] Such procedures are known as coroutines or semi-coroutines and they differ from normal procedures in that each time they are invoked from another coroutine they resume execution where they last left off. Their cooperative behavior is understandable in terms of an anthropomorphic model in which each coroutine is executed by a different person who simply goes to sleep when he resumes one of the other coroutines, but when his own coroutine is resumed again he awakens in the same state as before he went to sleep. Coroutines can be used to obtain the conceptual advantages of a multi-pass algorithm without the actual need for secondary storage and data format specifications usually implied by a literal implementation of the separate passes.[12] Especially compelling examples of the conceptual correctness of coroutines are to be found in Dahl.[11] The coroutines discussed here are never in simultaneous or interleaved execution so their correct behavior doesn't involve the deeper problems of mutual exclusion, deadlock, etc.

Conclusion

An attempt has been made to discuss various issues involved in the design of control for a programming language by relating these design issues to the goal of reducing "conceptual distance". A slight compromise to the strict structured control seems justifiable to obtain a more problem-oriented control. More research would be worthwhile in the area of "safe" iterations, parameter mechanisms and coroutines.

References

1. E.W. Dijkstra, "Notes on Structured Programming," in Structured Programming by O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare, 1972, Academic Press, London and New York.

2. C.A.R. Hoare, "Notes on Data Structuring," in Structured Programming (see 1).

3. N. Wirth, "The Programming Language PASCAL," Acta Informatica, Vol. 1, No. 1, pp. 35-63.

4. W.A. Wulf, "A Case Against the goto," Proc. ACM National Conference (1972), pp. 791-797.

5. W.A. Wulf, D.B. Russell and A.N. Haberman, "BLISS: A Language for Systems Programming," CACM, Dec. 1971, pp. 780-790.

6. C.T. Zahn, "A Control Statement for Natural Top-down Structured Programming," presented at Symposium on Programming Languages, Paris, 1974.

7. D.E. Knuth, "Structured Programming with goto Statements," ACM Computing Surveys, December 1974.

8. C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica, 1973, pp. 335-355.

9. C.A.R. Hoare, "Hints on Programming Language Design," Stanford University Computer Science Department Report No. CS-403, October 1973.

10. W.T. Hardgrave, "Positional versus Keyword Parameter Communication in Programming Languages," Report of the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, Virginia, September 1974.

11. O.J. Dahl, "Hierarchical Program Structures," in Structured Programming (see 1).

12. D.E. Knuth, "The Art of Computer Programming," Volume 1, Chapter 2, 1968.