Chapter 11 of Human Interaction With Computers; H T Smith & T R Green eds; Academic Press 1980.

Chapter 11

THE DESIGN AND USE OF CONVENTIONAL PROGRAMMING LANGUAGES

Michael Jackson

Contents

1.0 Introduction

Jean Sammet's Roster of Programming Languages for 1974-75 (Sammet 1976) includes 167 different programming languages which were in use in the United States at the end of 1975: a full list, including languages used chiefly outside the United States and languages which had fallen into disuse by the end of 1975, would perhaps have run to 500 languages. There are languages for special purposes: APT is for programming numerical control of machine tools; SIMSCRIPT is for discrete simulation; COGO is for civil engineering; CONNIVER is for artificial intelligence studies; COBOL is for administrative and business data processing. There are languages intended for general-purpose use: ALGOL 60, ALGOL 68, PASCAL, PL/I and SIMULA 67 are perhaps the best known of these.

The proliferation of programming languages has been an important factor in the spiralling costs of programming: just as most people speak only one natural language, so most programmers have learnt to use only one programming language; as a result, similar or even identical problems have been programmed many times in many places by many programmers. Even where two programmers use what is nominally the same language — say, FORTRAN — they are likely to use slightly different dialects of the language; the differences are often enough to prevent one programmer from using a program written by the other.

From time to time, efforts are made to devise a language which can be standardised and applied to a wide variety of problems in its standard form. PL/I is such a language: in its original conception it was to combine the best features of COBOL and FORTRAN, and would thus be well suited to both numerical scientific computation and business data processing; it was also to draw on the widely acknowledged elegance and power of ALGOL 60. PL/I was conceived in the early 1960s; in 1979, at the time of writing, the United States Department of Defense has placed contracts for the development of a new programming language (the "Ironman" project) appropriate for a wide range of applications in which the computer is an integral part of a complete military system.

Such efforts have not, in the past, met with any notable success. Opinions vary widely on the merits of PL/I; in spite of the high hopes at its conception, and the continued energetic support of IBM, few people today would regard PL/I as a definitive answer, even in the broadest terms, to the question of what a good general-purpose programming language should be. Informed opinion on the Ironman project is no more optimistic.

Numerical scientific computation and business data processing account for a very large proportion of all the programming work done in the past and being done today. Most of that programming is done in FORTRAN or COBOL, languages which are about 25 and 20 years old respectively. At least two reasons may be discerned for this apparent conservatism. First, and this is especially true of COBOL programs, a program is a capital asset; it is expensive to build, and the cost of building must be recouped over an extended period of years. An organisation which owns a million COBOL statements — say, 1000 programs of an average size of 1000 statements — can no more think of abandoning that investment than a nation can think of abandoning its road network or its telephone system. The resources required to replace it are simply not available, and even if they were, the cost replacement could hardly be justified. Second, it is becoming more widely recognised that a change of programming language can bring only very limited benefits. The defects which have been identified in existing programs and systems of programs will not be cured by rewriting them in a new language: they are defects of underlying structure attributable to the way the languages have been used rather than to the characteristics of the languages themselves.

In this chapter I would like to suggest that programming languages of the kind which are in wide general use today — and especially COBOL, FORTRAN and PL/I — suffer from a fundamental difficulty; they are trying to do two incompatible things at once. They are trying to allow the programmer to express what he wants the machine to do, and, at the same time, what order he thinks the machine should do it in. I am suggesting that both of these are necessary, but they cannot coexist in one language in the way they have done in the past. We need to separate the two concerns, forming two language classes, which we may call the Programming Languages and the Execution Languages. Only if this is done will we be able to rise above the difficulties which have been apparent, but apparently insoluble, for the past fifteen years.

2.0 Beginnings

Early computer programmers conceived of their task as one of directing the operation of the machine; indeed, in the incunabula of computer literature (for example, Mauchly (1947) and Alt (1948)) we usually find the programmer referred to as ‘the operator’. Given a problem, and some prior mathematical analysis, the programmer's task was to devise a sequence of machine operations which would solve that problem, encode the sequence of operations onto a tape, and present the tape as input to the machine. The program thus encoded was, in essence, the expression of the operator’s instructions to the machine: instead of pressing buttons, setting dials or pulling levers, the programmer wrote ‘commands’, ‘instructions’ or ‘orders’, which the machine then ‘obeyed’.

Since the programmer was concerned to direct the operation of the machine, step by step, he/she was required to understand and to take account of the detailed manner in which the machine worked. The exact representation of data, the use of the various machine registers, the absolute location of the program in core storage, the placing of subsequences of instructions on a drum to minimise rotational delays during execution, ordering instructions to take advantage of any available parallelism in the machine-all these were the programmer’s responsibility.

Later, symbolic assemblers and simple autocodes relieved the programmer of some of the more irritating clerical parts of the task; commands could be issued to the machine in a more palatable form than the machine's own order code, but they were essentially the same

commands and the programmer had essentially the same control over, and responsibility for, the action of the machine.

The development of 'high-level programming languages' such as FORTRAN and COBOL and their predecessors, was largely motivated by a recognition that the hardware machine, even when approached through the medium of a symbolic assembler, was not really suitable for solving either mathematical or data processing problems. The new languages transformed the machine, as it appeared to the programmer, into one with a more directly useful set of data types and operations. The hardware machine, with its highly specialised registers, accumulators and stores, and a wide repertoire of similarly specialised operations, was to be replaced by a software machine offering a smaller but more general repertoire better suited to the problems to be programmed. Programs in these high-level languages, while their texts might contain no fewer characters than the machine-language programs they replaced, consisted of fewer distinct statements and commands, more directly identifiable with steps in solving the problems as originally posed to the programmer. At the same time, structural mechanisms were formalised as declarations and calls of procedures, subroutines and functions, encouraging the programmer to abstract and generalise; it was hoped that large libraries of general-purpose procedures could be developed and used to good effect.

Of course, the underlying concept was still that of a machine to be instructed by the programmer. To be sure, the FORTRAN machine and the COBOL machine were more accessible and less esoteric than the 7090 and the 1401, but they were still definitely machines, and it was still the programmer's role to instruct them, to guide them step by step through the desired computation. The compilers were essentially translators, producing machine-code programs which followed very closely the pattern of the COBOL or FORTRAN program text. The declaration of an *INTEGER* in FORTRAN or a *PICTURE S9(8) COMPUTATIONAL* in COBOL produced something specific and predictable in the machine-code program, just as the declaration of a binary full word had done in the symbolic assembly language; each executable statement in the FORTRAN or COBOL text produced manifestly equivalent machine instructions in the corresponding place in the machine-code program.

2.1 Difficulties

In the early and middle 1960s, with the increasing availability of larger core stores and a consequent growth in the size of programs, a number of difficulties became apparent which were more fundamental than the unsuitability of the hardware data formats and instruction sets.

The foremost difficulty was complexity. The potential complexity of a program with an arbitrary control structure increases geometrically with the size of the program text; inevitably, most programmers found that they could not ensure the correctness of a program of 500 executable statements in which branching was permitted without constraint. Some programmers developed informal intuitive methods which allowed them, as individuals, to work effectively; some exercised capacious and surprisingly accurate memories in a triumph of brainpower over lack of method; others comforted themselves with the reflection that the inevitability of error was a delightful and impressive evidence of the sophisticated nature of their vocation.

The difficulties of complexity were exacerbated by the demands of program maintenance. The typical program was no longer an isolated object, to be run from time to time in response to ad hoc demands; it was a part of a system, to be run regularly in careful co-ordination with other programs of the system. The system itself was expected to have a lifetime of several years, and during that life-time its programs were subject to a succession of modifications as the system's specifications changed. A large data processing system, of 100,000 statements or more, is effectively a giant program running continuously for several years; not surprisingly, changing one of the constituent programs of such a system is difficult and error-prone. As more such systems came into existence, the ratio of

program maintenance to original programming work rose steadily; today the ratio in data processing is commonly quoted as 2:1, 3:1 or even 4:1. The cost of program maintenance is determined largely by the quality of the original work done in the design and programming of the system, but in ways that are not well understood.

The difficulties of complexity and of program maintenance may be attributed, at least in part, to the unusual nature of programming languages as a construction medium. Programs are essentially homogeneous: the basic stuff of which they are built is provided by a relatively small set of elementary statements in the programming language, and this same stuff is used everywhere in the program to the exclusion of all else. Thus, a constraint which promotes modularity in physical artefacts such as motor cars and aeroplanes is absent. The programmer, unlike the mechanical engineer, is free to weld any part of a program to any other part, or, indeed, to mix up the parts in any desired way. It need not be considered that this part is made from steel, that from glass and a third from plastic — all can be merged and partitioned without constraint. Nor does the stuff of programs provide any constraint, such as the strength of physical materials or a required power-weight ratio. Any program, however poorly conceived, can be made to work by *ad hoc* expedients and patches. Since the structural form of the program is largely invisible to its buyers and users it is horribly easy — and tempting — to allow the quality of the design to sink to a standard which would not pass muster for a moment in a motor car or a bridge.

The ideas of modular programming emerged in response to these difficulties. The function of a program was to be decomposed into subfunctions, which in turn could be further decomposed, and so on for as many levels as necessary. For example, the function ‘Compute Pay’ could be decomposed into ‘Compute Gross’ and ‘Compute Net’; ‘Compute Gross’ into ‘Compute Basic Pay’ and ‘Compute Overtime Pay’; ‘Compute Net’ into ‘Compute Deductions’ and ‘Subtract Deductions from Gross’; ‘Compute Deductions’ into ‘Compute Tax’ and ‘Compute Other Deductions’; and so on. Each subfunction could then be programmed as a closed subroutine, preferably separately compiled, and the resulting program would be a hierarchical structure of these subroutines. By limiting the size of the individual module or subroutine complexity would be brought under control; by ensuring that each subfunction was programmed in a separate module maintenance would be localised; by recognising, at the design stage, generally useful subfunctions, a library of modules would be created which could reduce the amount of new work required for each successive program.

These high hopes were not realised except to a very limited extent. It became clear that the task of the designer, the decomposition into subfunctions, was crucial to the success of the program: if it was done poorly, the mechanics of modularisation would actually make the program worse rather than better, separating what should have been together and incurring large overhead costs in the space and time needed for the program’s execution.

The chief residual effects of modular programming were a widespread acknowledgement of the importance of program design as a subject in itself, and an almost universal conviction that the subroutine call was the primary, or even the only, structural device. There was some debate about the value of separate compilation, and some COBOL compilers were slow to acquire *CALL* and *ENTRY* statements; but it was clear that the subroutine call was the natural and obvious means of connection between a function and its subfunctions.

This emphasis on the subroutine call, like functional decomposition itself, was also an organic development of the earliest view of the programmer’s task. The program was a sequence of instructions to the machine; if a particular instruction, such as ‘divide’, was not in the machine’s repertoire, it could be readily simulated by a subroutine call. In the same way, more elaborate instructions, such as ‘calculate gross pay’, or ‘update master record from transaction record’, could be simulated too. Programmers would still instruct the machine exactly: but were not able to provide themselves with a variety of machines at various levels of the subroutine hierarchy, each machine having a carefully crafted instruction repertoire of their own devising.

2.2 The GO-TO Controversy

The next substantial contribution to the conceptual basis of conventional programming was provided by the seminal work of Dijkstra (1968; 1970; 1972) on Structured Programming. However, before continuing with that theme, a major digression is necessary. Structured Programming contains two main strands: one is the adoption of an explicit design concept, stepwise refinement; the other, which gave rise to the celebrated GO-TO controversy and thus makes our digression necessary, is the strong recommendation of a sequencing discipline to limit patterns of control flow.

A programmer who makes free use of GO-TO statements, permitting a jump from any point to any other point in a program, is in serious danger of constructing programs whose action he/she cannot predict. Such a free use of GO-TO statements was a usual ingredient of programming technique, and a major contributor to the complexity and obscurity of the resulting programs. Dijkstra (1968) recommended that control flow patterns should be limited to the three, now classical, constructs of 'concatenation', 'repetition' and 'selection'; in this way the dynamic behaviour of a program (that is, the computations it can evoke) is simply and readily deducible from the static text. These points are illustrated by Figures 1 & 3 of Green's chapter in this volume. It had been shown by Bohm and Jacopini (1966) that any proper flowchart (that is, a flowchart with one entry and one exit and all connected) could be decomposed into a hierarchy of these constructs, although in general the decomposition would require duplication of some parts of the flowchart and the introduction of some new Boolean variables: the limitation on control flow patterns recommended by Dijkstra was not therefore a limitation on the programs that could be written.

Unfortunately, a catchy and easily remembered headline "GO-TO Statement Considered Harmful" was added to Dijkstra's letter to the Communications of the ACM (1968). As his ideas spread, they became diluted and oversimplified; it seemed attractive to encapsulate the whole matter in a single-minded ban on the GO-TO statement, presenting a complex intellectual issue as a readily understood slogan. Control flow is a complex issue for at least two reasons. First, because it is possible to conceal sequencing by implementing it in program variables instead of in the explicitly sequenced executable text: at the limit, any program can be expressed by the interpretive scheme:

```
begin instruction counter := 1;
  do while instruction (instructioncounter) ≠ 'halt'
    execute instruction (instructioncounter);
    set next value of instructioncounter;
  od
  halt;
end
```

The true control flow of a program is not therefore always evident. Second, because it is not clear that concatenation, repetition and selection are an entirely adequate set of constructs: several writers, including Zahn (1974) and Jackson (1975), have proposed further constructs.

To identify Structured Programming with the avoidance of the GO-TO statement is simplistic, and traduces the depth and importance of the ideas involved. But such an identification has been widely made, and has had an eccentric effect on the comparative evaluation of programming languages. Attention has been concentrated on a minor issue of superficial syntax, on whether a particular language allows repetition and selection to be expressed without resorting to the condemned GO-TO, and has thus been drawn away from more significant questions. The point is well illustrated by the widely held opinion that FORTRAN is a very bad language for Structured Programming, while ALGOL 60 is very good for the same purpose. FORTRAN provides anomalous and idiosyncratic constructs for repetition and selection; so anomalous are they that it is generally held, with considerable justification, that the only practicable way of writing 'structured programs' in FORTRAN is to simulate the repetition and selection constructs by the use of GO-TO

statements and labels. ALGOL 60, on the other hand, provides something more appropriate. Ergo, ALGOL 60 is good, and FORTRAN is bad. But neither of these languages provides for any kind of non-elementary data construct than the homogeneous array, all of whose elements are of the same type. This is a much more serious defect, which both languages exhibit: whereas the lack of repetition and selection in logic flow can be readily overcome by simulation of the constructs with GO-TO statements and labels, the lack of data structures cannot be overcome at all easily. FORTRAN programmers are well accustomed to the unpleasant dilemma between passing a large number of parameters to a subroutine, each referring to a single variable or array, and placing the variables in *COMMON* storage, thus bypassing the parameterisation mechanism with a consequent vulnerability to many types of error. The better solution would be to form the variables into one or more data structures, of the kind provided in COBOL and PL/I, and also in later ALGOL-derived languages such as PASCAL, and to pass references to those data structures as parameters to the subroutine. But neither FORTRAN nor ALGOL 60 provide such data structures.

In a similar vein, and for similar, reasons, PL/I is judged better than COBOL. Proponents of COBOL and FORTRAN rush to equip their favoured languages with 'structured programming facilities' which will allow textual nesting of reasonably sanitary repetition and selection constructs. It is interesting to observe that the new facilities are often to be provided by precompilers and preprocessors which detect the new constructs and translate them into the standard language which the standard compiler can handle. Such precompilers and preprocessors are relatively simple and cheap to build, allowing experiments in programming languages to be made without incurring the cost of a full compiler and a full language definition. In programming language design, the age of optimism has passed away.

2.3 Levels of Abstraction

The other strand of Structured Programming is an explicit concept of program design: design by stepwise refinement. (The same notion, in a slightly coarsened form, is often known as 'top-down design'.) Using stepwise refinement, the programmer starts with a very simple statement of the algorithm in terms of very high-level actions and data objects; at each step the actions and data objects are refined until eventually they are expressed in terms of actions and data objects which are available in the repertoire of the programming language being used. The progress from step to step may be considered to be a progress from the more abstract to the more concrete, from the programming of an idealised conceptual machine to the programming of the machine which is actually available. In this way, the programmer is encouraged to see a relation between an abstract conception of the problem at the first step, and the concrete implementation of its solution at the last: the purpose of the later steps is to give effect to the earlier. The point is well expressed in a remark by Dijkstra in a recent book (1976):

> "It used to be the program's purpose to instruct our computers; it became the computer's purpose to execute our programs."

In the first steps of stepwise refinement, the programmer was to become free of the task of instructing the computer; only in the last step need that responsibility be considered directly.

The change of viewpoint implied is of fundamental significance; but the new freedom for the programmer, the freedom from the responsibility to direct the step-by-step operation of the machine, is more apparent than real. The programmer is free to postpone questions of representation, both of data objects and of operations on those data objects; but is not free to avoid them altogether, since in the final step of stepwise refinement representations must be chosen in the programming language being used. More important, the programmers are not freed at all from the need to decide the exact sequencing of the machine operations. Even in the very first step they commit themselves to the exact sequencing of the high-level or abstract operations. For example, in solving the problem of

printing the prime numbers less than 1000, the programmer must choose whether the first step is to be

```
begin
  create table of primes;
  print table of primes
end
```

or

```
for n := 1 step 1 until 1000
  if n is prime then print n
  fi
```

or

```
begin
  n := 2;
  while n < 1000
    do print n; generate next prime n
    od
end
```

or some other possible sequence. Evidently, this choice at the first step predetermines the sequencing of the final program to a large extent; further, it affects the representation of the primes themselves; the table of primes used in the first choice could be conveniently represented by a string of 1000 bits, but such a representation would hardly come naturally to a programmer who had chosen the second or third possibility.

3.0 Freedom from the Machine

Freeing the programmer from the tyranny of the machine had been a recognised objective from the earliest days of symbolic assemblers. The machine dealt in absolute storage addresses, but the users of a symbolic assembler were free to deal in names of their own choosing for both data and instructions. The user of a language like COBOL or FORTRAN was invited to think in terms of data types judged suitable to the problem, and appropriate operations on objects of those types. The generated machine instructions were not, in principle, of concern, although programmers could, and often would, make it their business to know what they were: the action of the compiler in generating the object machine program was entirely predictable.

More significantly, the programmer was invited to give up control of certain aspects of the generated object program which then became effectively unpredictable. In machines which used program-addressable registers for arithmetic or for addressing core storage, the allocation and use of the registers was left to the compiler, and the programmer could do nothing to affect it. In IBM Virtual Storage systems the pagination of the program was outside the programmer’s control.

The justification for removing these matters from the scope of the programmer’s consideration was, of course, that this thereby enabled attention to be concentrated on the problem rather than on the details of the machine’s execution of the program. The programmer was using a ‘problem-oriented' language rather than a ‘machine-oriented’ language. Correct and clear solution of the problem was more important than extreme efficiency in the use of the machine.

Many programmers, with varying degrees of justification, fought against this trend. Unwilling to abandon control over the machine, they resorted to a number of expedients. One expedient was to discover the rules by which the compiler generates machine instructions from statements of the programming language, and then to choose programming language statements with a view to efficiency of the generated machine instructions.

For example, a compiler might generate a call to an exponentiation subroutine when it encounters a statement such as

```
x:=y**z;
```

aware of this, and of the relative inefficiency of such a call, a programmer wanting to write

```
x:=y**2
```

might instead choose to write

```
x:=y*y
```

which would generate a simple multiplication instruction instead of the subroutine call. In the same vein the programmer might write

```
x:=y+y
```

in place of

```
x:=2*y
```

in order to generate an addition instead of a multiplication instruction. Obviously, such an expedient depends heavily on an accurate knowledge of the code generation part of the compiler, including any optimisation; it is easy to imagine perfectly sensible compilers for which the preferred statements shown above would give rise to less efficient, rather than more efficient, machine code.

On a grander scale, we see the same technique advocated for efficient use of IBM Virtual Storage systems. A virtual storage system is intended to permit the programmer to write programs whose instructions and internal variables occupy more main storage than is physically available in the machine; the virtual storage system allows some of the instructions and internal variables to be held in secondary storage, such as disk or drum, and to be brought into main storage when they are required. It is intended that this swapping of 'pages' between main and secondary storage should be invisible to the programmer, who should be able to program as if the machine's main storage were as large as desired. Obviously, a program which requires many pages to be swapped during execution will be less efficient than one which requires fewer; the programmer is therefore tempted to discover how the compiler allocates storage in a machine code program and how the paging system operates, so that advantage can be taken of this knowledge in planning the program. Rogers (1975) suggests such guidelines as:

- Separate unused code and data space from code that is frequently used.
- Seek algorithms or techniques that use small data areas.
- Store data as closely as possible to other data that are to be used at the same time.
- Data should be referenced in the order in which they are stored, if possible, especially for large data aggregates occupying several pages.
- If possible, separate read-only data from areas that are to be changed.
- Avoid implied FORTRAN DO loops in I/O statements because they cause repeated return to the calling program.

Some of these guidelines require the programmer to make a particular choice between apparently equivalent alternatives in the programming language; some (such as the second and fourth) will affect his program substantially. Essentially, the programmer is invited to circumvent the efforts of the language designer and of the virtual storage system designer to relieve him of the need to consider storage allocation at the machine level.

Even more determined unwillingness to lose control over the machine is shown by another widely used expedient. In IBM S/370 COBOL, the machine format of a *COMPUTATIONAL-3* item is packed decimal, in which two decimal digits are encoded in each byte, except for the rightmost byte which contains the least significant digit and the sign.

Where the sign is not required, and the variable has an even number of digits, some space can be saved by creating a new variable type: unsigned packed decimal. The following illustration shows the technique:

```
02 TEN-TIMES-AND-SIGN PICTURE S9(5) COMPUTATIONAL-3.
02 FILLER REDEFINES TEN-TIMES-AND-SIGN.
 03 UNSIGNED-PACKED-DECIMAL PICTURE XX.
 03 FILLER PICTURE X.
```

The technique depends, of course, on adding 10 to the item TEN-TIMES-AND-SIGN whenever 1 is to be added to the item UNSIGNEDPACKED-DECIMAL. The resulting program will be obscure to the point of opacity: the inevitable disadvantages should be regarded as a measure, not of the programmer's folly, but of the importance in the eyes of the individual of retaining control over the machine code program.

3.1 Embracing the Machine

A more simple and direct approach to the problem of obtaining efficient machine code from programs written in high-level languages is to introduce overtly machine-oriented elements into the languages themselves. This approach is, of course, in conflict with the trend towards problem-oriented programming; it is an avowed reversion to the idea, at least in part, that the purpose of the program is to instruct the machine; but it has been widely adopted.

An obvious example is the provision, in the programming language, for explicit specification of machine representation of variables. COBOL permits the programmer to specify *COMPUTATIONAL, COMPUTATIONAL-1, COMPUTATIONAL-2, COMPUTATIONAL-3,* etc., for numeric variables: these are specifications of ‘implementer-defined meaning’, giving in the case of IBM S/370 COBOL, binary half-word or full-word, floating-point, double-precision floating-point, packed decimal, etc., representations. PL/I provides purely machine-oriented types for numeric variables, in the form *FIXED* and *FLOAT, BINARY* and *DECIMAL:* no nonsense here about problem-oriented data types.

Another example, showing a regression from problem-oriented to machine-oriented language, is the *INDEX* feature of COBOL. Originally, COBOL provided subscripting, much after the manner of FORTRAN, although subscripts could not be expressions. The compilers commonly available tended to generate very inefficient machine code for subscripted references; the inefficiency was worse in COBOL, where subscripted variables were likely to be of arbitrary size, than in FORTRAN, where they were likely to be represented by single machine words. The *INDEX* feature was introduced to make it easier for the compiler to generate acceptably efficient machine code; the price paid by the programmer was a new set of operations and declarations for *INDEX* variables themselves, and the arbitrary restriction that an *INDEX* variable, unlike a subscript, may refer to the elements of only one array.

The most dramatic example, however, of machine-orientation in programming languages is so deeply embedded in much of our thinking that it goes largely unnoticed. The machine conventionally presented to the programmer is a single processor: there is only one processing unit, and only one instruction address register. Since it follows that the machine can be doing only one thing at one time, the instructions of the program must be executed in some sequence or other. Conventional programming languages, as commonly used, require programmers to specify the exact sequence of instruction execution, whether or not they wish to do so.

I will argue below that this over-specification of sequencing is harmful. It necessitates a multiplicity of sequencing mechanisms; it burdens the programmer with a task which often need not be carried out to solve the problem; it is a fruitful source of error; and it obscures the simplicity of some problems by confusing the problem itself with the mechanics of executing its solution.

3.2 Resolving the Conflict

There are thus two conflicting pressures in programming language design. On the one hand, there is the recognition that the ideal programming language would be oriented towards the problems which the programmer is trying to solve, and would not require the burden of considering too closely the characteristics and operation of the machine. On the other hand, we cannot afford to abandon our concern with the machine. Neither the general-purpose compiler nor the general-purpose operating system is capable of producing the efficient execution we desire and need; we must still issue explicit instructions about the representation of data, the sequencing of machine operations, and other characteristics of program execution.

The only resolution of the conflict must lie in ceasing to demand these conflicting services from our programming languages; we should recognise the need for a clear separation of the problem-oriented from the machine-oriented concerns, and we should make exactly that separation in our languages. Any language would be split into a pair of complementary languages: the Programming Language proper, in which we write our solution to the problem, and the Execution Language, in which we specify how that solution is to be compiled and executed to obtain the necessary efficiency.

The vestiges of such a separation are already visible in COBOL: the COBOL Environment Division was intended to provide a separate place for the programmer to specify certain machine-oriented requirements, especially in the matter of physical file formats. The idea is that machine-oriented characteristics are to be specified in the Environment Division, while problem-oriented characteristics are to be specified in the Data Division. Thus, to define a serial file of 80-character card images, the programmer writes

```
DATA DIVISION.
FILE SECTION.
FD CDFILE, DATA RECORDS ARE CARD-IMAGE.
01 CARD-IMAGE PICTURE X(80).
```

and to specify that the file is to be physically held on magnetic tape the following is written, separately,

```
...
ENVIRONMENT DIVISION.
...
    SELECT CDFILE, ASSIGN TO MAGNETIC-TAPE-UNIT.
```

in which the *SELECT* sentence specifies the machine-oriented characteristics, while the *FD* (File Definition) entry in the Data Division specifies the problem-oriented structure of the data records. Unfortunately, the allocation of further specifications to the *SELECT* sentence or *FD* is bizarre: for example, physical blocking of the file, which is invisible to the program, is specified in the *FD* entry; the name of the record-key item, for a file to be accessed by key, is specified in the *SELECT* sentence. A decision has recently been taken to reallocate these and oth[e]r specifications to their proper place.

The pattern of programming activity which would emerge from such a separation between the Programming Language and the Execution Language is clearly one of two ordered steps:

1: The problem is solved in the Programming Language, without consideration of the execution characteristics of the solution.

2: The Execution Language is used to specify the execution characteristics, indicating how the Programming Language program should be compiled and executed to obtain acceptable efficiency.

The statements in the Execution Language would refer to the Programming Language program text, but would not modify it in any way: they would only direct the operation of the compiler and, perhaps, of the operating system.

Such a scheme could be considered a full satisfaction of P. Landin's prescription: "most programmers start by writing a program which they know to be efficient, and then transform it during debugging by transformations which they hope will make it correct while preserving efficiency; they ought instead to start by writing a program which they know to be correct, and then transform it by transformations which they know will preserve correctness and hope will make it efficient."

4.0 Separating the Problem from the Machine

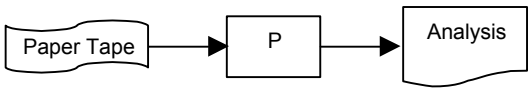
4.1 Process Scheduling

One of the aspects of excessive machine-orientation mentioned above (in section 2.2) is the requirement that the programmer specify, in the program, the exact sequence of machine operations to be executed by the single processor which the programming language assumes. In this section, we will illustrate how that machine-oriented concern may be isolated from the problem solution and relegated to the Execution Language.

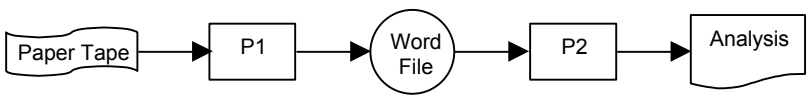
As illustration, we take the following example problem, derived from a problem originally posed by Henderson & Snowdon (1972) and further treated by Ledgard (1973) and by Gerhart & Yelowitz (1976).

> "An input file on paper tape contains the texts of a number of telegrams. The tape is accessed by a 'read block' operation, which reads a variable-length character string delimited by a terminal EOB character; the size of a block cannot exceed 100 characters, excluding the EOB character. Each block contains a number of words, separated by space characters; there may be one or more spaces between adjacent words, and at the beginning and end of a block there may (but need not) be one or more additional spaces. Each telegram consists of a number of words followed by the word 'ZZZZ'; the set of telegrams is followed by a null telegram, consisting only of the 'ZZZZ' word. There is no special relationship between telegrams and blocks: a telegram may begin and end anywhere in a block, and may span several blocks; several telegrams may share a block. There is a special end-of-file block, recognisable as producing the result EOF from a 'read block' operation; no word can occur after the 'ZZZZ' word of the null telegram and before the EOF marker block. The telegrams are to be analyzed, and a report printed showing for each telegram the number of words it contains and the ordinal number of the telegram. 'ZZZZ' does not count as a word for purposes of the analysis, nor does the null telegram count as a telegram."

An approach to this problem, as given in Jackson (1975), may be based on a consideration of the data structures of the input and output files, seen essentially as grammars of regular expressions. (Figure 1 illustrates this approach.) It is then clear that there is a 'structure clash', and that the solution to the problem must be constructed in two parts: one part dissects the paper tape file into a stream of words, while the other reconstructs the stream of words into a file of telegrams. We are thus led to view the program not as one process, but as two; not as



but as



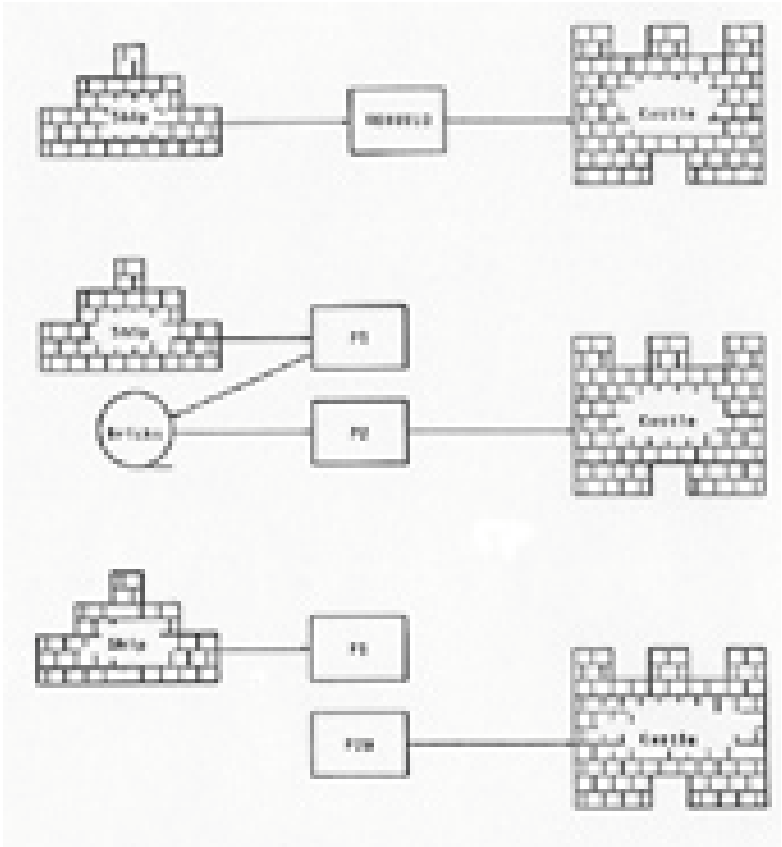


Fig. 1. A structure clash and its resolution. A Lego model of a ship is to be rebuilt as a castle, but while the ship consists of a hull, a superstructure and a funnel, the castle has a very different structure. Obviously the solution is to create two different programs, one to deal with each structure. These are written in the Programming Language; how are they to be expressed in the Execution Language? One way would be to use P1 to dismantle the ship into a pile of bricks and then use P2 to build a castle from the pile, but in real terms this would he very inefficient because the 'bricks file' would have to be completely written before the first record could be read. But instead we could 'invert' one program, say P2, into a subroutine of the other, say P1. We will call the new program P2B. PI calls P2B to dispose of a brick of the brick file, which it does by adding it to the castle. It turns out that P28 is effectively identical to P2, except for a minor mechanical transformation at the coding level. There is no difficulty in devising suitable implementations in languages like COBOL and PL/I. Moreover there is no reason why a compiler should not be capable of compiling P2 or P2B indifferently from the same source text.

This decomposition into two processes can also be justified by the need to build a correct model of the reality which is the subject matter of the computation. In the problem environment, there are evidently two distinct and independent processes: on the one hand, there is the process of receiving telegram texts from customers and counting the words in order to make a correct charge; on the other hand, there is the process of punching these texts into paper tape, block by block. The telegram process is iterative by telegrams, while the punching process is iterative by paper tape blocks; the only connection between them is that the punch operator is required to punch into paper tape the words of the telegrams in the order in which they occur in the telegram texts. It is therefore not merely appropriate but necessary for correct design that the solution should consist of two processes mediated by a serial stream of words.

The program texts for the resulting programs, P1 and P2, are as follows:

```
P1: begin
     open ptape; read ptape;
     open wordfile;
     do while not EOF
        ps := 1;
        do while ptch(ps) ≠ EOB
           if ptch(ps) = space
           then ps := ps + 1
           else
              begin
                 ws := 1;
                 do while ptch(ps) ≠ EOB
                          and ptch(ps) ≠ space
                    wch (ws) := ptch(ps);
                    s := ps + 1; ws := ws + 1
                 od
                 write word
              end
        od
        read ptape
     od
     close ptape; close wordfile
    end
```

```
P2: begin
     open wordfile; read wordfile; open analysis;
     print `TELEGRAMS ANALYSIS';
     tn:=0;
     do while word ≠ 'ZZZZ'
        nw:=0; tn:=tn+1; nw:=nw+1;
        read wordfile;
        do while word = 'ZZZZ'
           nw := nw + 1;
           read wordfile
        od
        print 'TELEGRAM NO', tn,
              'CONTAINS' nw, 'WORDS';
        read wordfile
     od
     read wordfile;
     print 'END ANALYSIS';
     close wordfile;
     close analysis
    end
```

The problem solution is now effectively complete in the Programming Language. However, it still remains to specify in the Execution Language the scheduling of the two processes so that they may share a single CPU.

There are, of course, several choices available to us. We could run P1 to completion before starting P2; the controlling algorithm within the operating system would then have the simple form:

```
begin execute P1 to completion;
      execute P2 to completion
end
```

and the ‘read’ and ‘write’ operations in P2 and P 1 respectively would refer to a sequential file on backing store such as magnetic disk or tape. Or we could swap between PI and P2 on each word, when there would be a controlling algorithm of the form:

```
do while not finished
execute P1 until it produces a word;
  execute P2 until it needs the next word
od
```

and the ‘write’ and ‘read’ operations would merely be suspensions of the active process and resumptions of its partner.

These are not the only two available choices; but they are noteworthy because they allow the scheduling to be fully determined at compile time.

The first choice can be implemented by the compilation of the ‘read’ and ‘write’ operations into appropriate invocations of input-output procedures, together with the generation of a suitable declaration of the wordfile and the ordered invocation of PI and P2. The second choice can be implemented by the compilation of P1 and P2 as coroutines, with the ‘read’ and ‘write’ operations generated as ‘resume P1’ and ‘resume P2’ respectively; or by ‘program inversion’ of either P 1 of P2 (Jackson 1975) making one of the pair into a subroutine of the other (a semi-coroutine).

There are other, more elaborate choices, such as the introduction of a bounded set of buffers between P1 and P2, which leave the scheduling to be partly determined at execution time.

Now, whichever choice we make, and express in the Execution Language, we are surely entitled to expect that the program texts will remain unchanged in the Programming Language: the computation carried out by each of the two processes is unchanged, and the results of the two processes taken together are unchanged. There seems to be no good reason why our choice of scheduling should affect the texts of the scheduled processes, P1 and P2. To put the same point the other way round, there is no good reason why the texts should determine the process scheduling.

Unfortunately, however, in the commonly used programming languages the scheduling of the two processes is expressed directly by the way their texts are written. The variety of the possible program texts that can result is a serious barrier to simplicity and understanding in programming. The fundamental communication between the two processes, the writing and reading of the serial file wordfile, is obscured by the specification of a scheduling scheme which is entirely irrelevant to either process taken alone. It is as if we were required to use completely different syntax for the same program according to whether it is run under a multi-programming or a uni-programming operating system. In effect, the programmer is forced to overspecify the sequencing of the program: a program has been designed consisting of two processes; the communication between them has been specified by means of the wordfile; the programming language forces the specification, in the most obscure and inconvenient way, of the scheduling of the two processes on a single-processor machine.

4.2 Procedures and Processes

In the preceding example, the fundamental structuring device is decomposition into processes communicating by serial file operations, by passing streams of records. The procedure invocation plays no part at all, except that we may, under duress, press it into service as an implementation of a particular scheduling scheme. But we would prefer to use a programming language which (lid not place us under such duress, and to relegate the scheduling concerns to the Execution Language.

The process. rather than the procedure, is appropriate as a fundamental structural component because of its suitability as a modelling medium. In a very large class of problem we are concerned with a reality as the subject matter of the computation. which can usefully be regarded as peopled by individual, independently active, entities. For example, in business data processing we may be concerned with customers, with suppliers, with employees, with products; in operating systems we are concerned with user jobs, with files, with hardware devices, with periods of usage of resources such as main storage or

input-output channels; in airline reservations systems we are concerned with flights, with seats, with airports, with customers and journeys. It is attractive to model such realities in terms of processes, using one process for each individual entity: one process for each employee, one process for each hardware device, one process for each journey, and so on. (This kind of modelling is very closely akin to the ideas underlying the SIMULA class concept (Dahl, 1972) and more recent work on abstract data types (Guttag, 1976): an individual object in the program is present for each individual entity in the modelled reality.)

To illustrate the point from the preceding example, the essential sequentiality of the paper tape is represented by the text of P1, while the essential sequentiality of the telegrams is represented by the text of P2: we may, as it were, follow the lifetime of the paper tape punch operator by following the text pointer of P1, and the lifetime of the telegram clerk by the text pointer of P2. The text pointer of a single process, being unique, is a satisfactory representation of the progress through time of the individual entity. That is, if we know where the text pointer of P2 is currently pointing, then we know where the telegram clerk has reached in the supposed lifetime. We are easily able in the process text to direct that sequentiality which belongs to the reality we are modelling.

Another Example. The telegrams analysis example is particularly simple because its network of processes is linear. A slightly more complex kind of network is needed to handle a problem such as the following. The problem is taken from Dijkstra (1976), who in turn attributes it to R. W. Hamming.

The problem is to generate a sequence of numbers in increasing order. The values in the sequence are defined by three axioms:

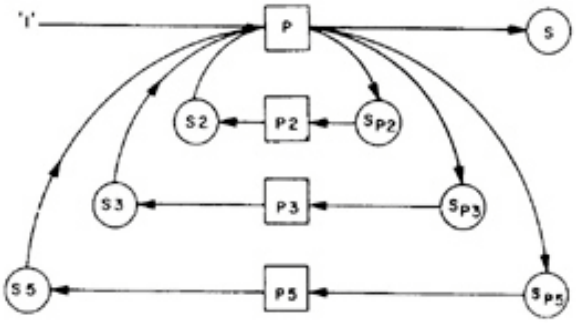
1. The value I is in the sequence.

2. If *x* is in the sequence, so are *2x*, *3x* and *5x*.

3. The sequence contains no other values than those which belong to it by virtue of axioms I and 2.

We may embody axiom 1 in a process P which produces the sequence S from the unique value 1:

We may embody axiom 2 by adding three further processes P2, P3 and P5, which generate respectively the values 2*x, 3*x and 5*x for each value in S:



SP2, SP3 and SP5 are simply copies of S: we prefer to avoid at this stage the difficulties of having one file input to (or output from) more than one process. P is now responsible for co-ordinating the files S2, S3 and S5 correctly, as well as for making the three copies of S.

Axiom 3, of course, we embody by refraining from adding further inputs to P.

The text of P is:

```
P: begin
     open, S, SP2, SP3, SP5;
     write 1 to S, SP2, SP3, SP5;
     open S2, S3, S5;
     read S2, S3, S5;
     n := min(S2, S3, S5);
    do while true
     write n to S, SP2, SP3, SP5;
     if S2 = n then read S2 fi;
     if S3 = n then read S3 fi;
     if S5 = n then read S5 fi;
     n := min(S2, S3, S5)
    od
   end
```

while the text of each of the Pi is:

```
Pi: begin
     open Si; open SPi; read SPi;
     do while true
        write i*SPi to Si;
        read SPi
     od
    end
```

The solution in the Programming Language may now be considered complete. It remains to determine, and to express in the Execution Language, the desired scheduling of the four processes.

The scheduling considerations are reasonably clear. First, the network contains cycles (P-SP2-P2-S2-P, etc.) and it is therefore not possible to run each process to completion (even if we had written terminal conditions for the do-loops) before starting another: we may not choose to run P first, producing SP2, and then, when P is complete, to run P2. Second, although the processes P, P2, P3 and P5 will necessarily be run with some degree of parallelism, it will not be possible to avoid buffering records. Clearly, P5 is producing its output file S5 'faster' than P3 is producing S3 — that is, from a given value *x* of S P5 produces *5*x*, which will occur later in S than the value *3*x* produced by P3; similarly, P3 produces S3 faster than P2 produces S2. At a general point in the execution of the whole program, therefore, there must be either records of SP5 which have been produced by P but not yet consumed by P5 or else records of S5 which have been produced by P5 but not yet consumed by P; and similar considerations apply to P3.

Scheduling of the network of processes is constrained, therefore, by more complex interconnections than we saw in the telegrams analysis problem; but it is still, in principle, possible to leave the scheduling to some suitable general-purpose operating system. In practice, such an operating system is unlikely to be available, and we are forced to consider explicitly questions of the scheduling of the processes and the associated buffering of records.

A reasonably simple implementation may be obtained by the following choices, to be expressed in the Execution Language:

- Only the first 1000 values of the sequence S are to be generated;
- The files SP2, SP3, SP5 are to be represented by arrays of integers, the writing and reading operations for those files being therefore represented by assignments to subscript variables and assignments to and from elements of the arrays;
- The same array will be used for all of the files SP2, SP3, SP5;
- P3 and P5 are to be activated only when P requires to read from S3 and S5 respectively: that is, only records of SP3 and SP5 need be buffered, not records of S3 and S5.

The reader may be sufficiently interested to work out the resulting program for himself, and to compare it with the program given in Dijkstra (1976). The same problem is also treated in Kahn & McQueen (1976), where there is a very illuminating discussion of the programming of process networks.

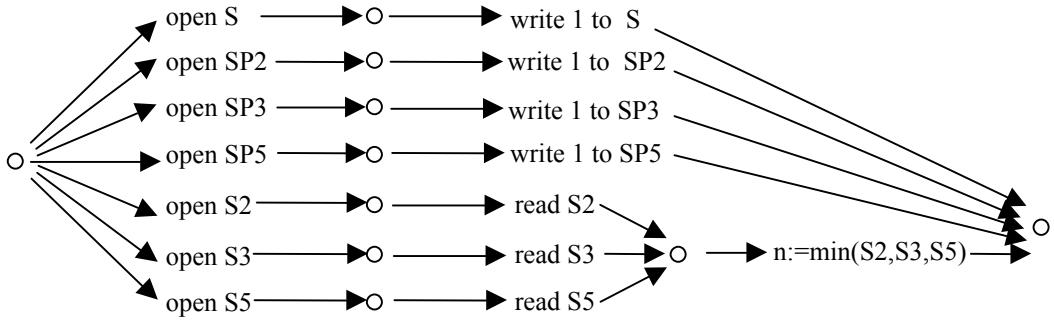
4.3 Microscopic Sequentiality

If, in the preceding example, we had been a little more careless (or a little less lucky), we could have created a deadlock. Suppose that we had written P as:

```
P: begin
     open S, SP2, SP3, SP5;
     open S2, S3, S5;
     read S2, S3, S5;
     write 1 to S, SP2, SP3, SP5;
     ...
```

This would have been perfectly satisfactory from the point of view of P alone. However, the system would have been deadlocked: P cannot proceed beyond the 'read S2' operation until P2 writes the first record of S2; but, as inspection of the text of P2 clearly shows, P2 cannot write the first record of S2 until it has read the first record of SP2, which P in turn cannot yet write.

The deadlock thus arising would be a consequence of the overspecification of sequentiality in P. The initial operations of P are only partially ordered: we may represent the partial ordering by the precedence graph:



It appears at first sight desirable to equip the Programming Language with a means for expressing such partial orderings, to avoid the danger of overspecifying sequentiality; but the means would apparently be cumbersome and difficult to use.

The deadlock threatened in the preceding example can be avoided by a relatively simple rule for ordering operations within a process: the rule is to execute all 'write' operations as early as possible, and to postpone all 'read' operations to as late as possible. It may be practicable for the compiler to reorder operations automatically to satisfy this rule, or it may require some interactive involvement on the part of the programmer, using the Execution Language.

5.0 Summary

Evidently, the two examples given have done little more than break the ice: we have aimed only to use the notion of a system as a network of processes to illustrate the separation of sequencing which is essential to solve the problem, from sequencing which is convenient or necessary for efficient execution of that solution. And this separation, in turn, is only one illustration — although arguably the most important — of the separation of problem-oriented from machine-oriented concerns.

The idea of the programmer's task as control of the detailed operation of the machine is deeply ingrained in today's conventional programming languages, and in the way we have

become accustomed to think about programming. Indeed, a programmer internationally renowned for his skill, asked at a conference on Structured Programming to explain his success, began his reply with the words

> "When I am programming, 1 carry in my mind a picture of the machine registers, the storage areas I have declared, and the values they currently hold..." (Baker, 1974).

Also deeply ingrained is the idea that the Programming Language is the medium for the exercise of that detailed control Knuth (1974) quotes with approval a remark of C.A.R. Hoare to the effect that "an optimising compiler should be able to explain its optimisations in the source language". Some interesting work on control structures (de Millo & Eisenstadt, 1976) has been published under the title "Can structured programs efficiently simulate GO-TO programs?". Should we not similarly ask whether ALGOL 68 can efficiently simulate the IBM System/370?

I have argued that these deeply ingrained ideas are harmful to the development of programming languages, and that the time has come for us to consider relegating to a separate Execution Language those concerns which are fundamentally related not to the correctness of a solution but to the characteristics of its execution. We can already see, in some of the work which has been done in program transformation, an explicit recognition of the harm which execution considerations can do to the practice of programming. For example, Burstall & Darlington (1975) write:

> "It is perhaps surprising to notice that even in the ratified language of purely recursive programs there is a sharp contrast between those written for maximal clarity and those written for tolerable efficiency. ... We are interested in starting with programs having an extremely simple structure and only later introducing the complications which we usually take for granted even in high level language programs."

But Burstall and Darlington are considering transformations of programs into different programs written in the same language; so too are some other researchers in the same field, such Arsac (1978). They, therefore, leave open the possibility of the programmer exerting himself to produce the transformed program directly in the first stage of his programming activity: by contrast, I have argued that the Programming Language should be purged of those elements which make such an exertion necessary today and would otherwise continue to make it possible tomorrow. This, perhaps, is the ultimate realisation of that separation of concerns which lies behind many, if not most, of the major advances already made in conventional programming technique and language.

References

Alt, F.L. (1948). A Bell Telephone Laboratories computing machine. MTAC 3, 1-13, 69-84. Reprinted in The Origins of Digital Computers (Ed.) B. Randell. Springer Verlag, New York. 1975. (2nd Edition.)

Arsac, J. (1978). An Interactive Program Manipulation System for Non-naive Users. Report No. 78-10, Laboratoire Informatique Theorique et Programmation, University of Paris, 2, Place Jussieu, 75221 Paris Cedex 05.

Baker, F.T. (1974). Infotech Conference on Structured Programming. London 1974 (quoted from memory).

Bohm, C. & Jacopini, G. (1966). Flow Diagrams, Turing machines, and languages with only two formation rules. Comm. ACM 9, 5, 366-371.

Burstall, R.M. & Darlington, J. (1975). Some Transformations for Developing Recursive Programs. Proc International Conference on Reliable Software. Sigplan Notices 10, 6, 465-472.

Dahl, 0.1. (1972). Hierarchical Program Structures. In Structured Programming (Eds.) O-J Dahl, E.W. Dijkstra and C.A.R. Hoare. Academic Press, New York. Dijkstra, E.W. (1968). Go-To statement considered harmful. Comm. ACM 11, 3, 147-148, 538, 541.

Dijkstra, E.W. (1970). Structured Programming. In Software Engineering Techniques (Ed.) J.N. Buxton, & B. Randell. NATO Scientific Affairs.

Dijkstra, E.W. (1972). Notes on Structured Programming. In Structured Programming. O-J Dahl, E.W. Dijkstra & C.A.R. Hoare. Academic Press, New York. Dijkstra, E.W. (1976). A Discipline of Programming. Prentice-Hall, New Jersey. 201.

Gerhart, S. & Yelowitz, L. (1976). Observations of Fallibility in Applications of Modern Programming Methodologies; IEEE Trans. on Software Engineering SE-2, 3, 195-207.

Guttag, J.V., Horowitz, E., & Musser, D.R. (1976). *Abstract Data Types and Software Validation.* University of Southern California, Los Angeles. ISI/RR-76-48. Henderson, P. & Snowdon, R. (1972). An Experiment in Structured Programming. *Bit, 12,* 38-53.

Hoare, C.A.R. (1974). *Hints for Programming Language Design.* Computer Science Report STAN-CS-74-403; Stanford University.

Jackson, M.A. (1975). Principles of Program Design. Academic Press, London.

Kahn, G. & MacQueen, R. (1976). Coroutines and Networks of Parallel Processes. IRIA Rapport de Recherche 202.

Knuth, D.E. (1974). Structured Programming with Go-To Statements. *ACM Comp. Surveys,* 6, 4, 261-301.

Ledgard, H.F. (1973). The Case of Structured Programming. Bit, 13, 45-57. Mauchly, J.W. (1947). Preparation of problems for EDVAC-type machines In *Annals of the Computation Laboratory of Harvard University,* 16, 203-207. Reprinted in *The Origins of Digital Computers (Ed.)* B. Randell. Springer Verlag, New York. 2nd Edition.

Millo, R.A. de, Eisenstadt, S.C. & Lipton, R.J. (1976). Can Structured Programs be Efficient? *Sigplan Notices, 11,* 10, 10-18.

Rogers, J.G. (1975). Structured Programming for Virtual Storage Systems. *IBM Systems Journal, 14,* 4, 385-406.

Sammet, J.E. (1976). Roster of Programming Languages for 1974-75. *Comm. ACM,* 19, 12, 655-669.

Zahn, C.T. (1974). A control statement for natural top-down structured programming. In Proc. *Symposium on Programming Languages.* Paris.