NESTED INTERPRETERS and SYSTEM STRUCTURE

by
Michael J. Manthey

DAIMI PB-51
September 1975

Matematisk Institut Aarhus Universitet
DATALOGISK AFDELING
Ny Munkegade - 8000 Aarhus C - Danmark
Tlf. 06-12 83 55

Nested Interpreters and System Structure

by
Michael J. Manthey

Abstract

The paper treats the problem of nests of interpreters i.e. multiple levels of interpretation of programs. The goal of the paper is to expose a control structure which models the important characteristics of such nests. Using a comparison with the Burroughs B6700 philosophy, a more general system model is introduced. There are two major results - a new type of control structure, and the conclusion that generalized nests perforce require retained environments.

Keywords: interpretation, microprogramming, retention, higher level languages, machine architecture, multiprogramming, control structures.

CR categories: 4.12, 4.13, 4.22, 4.32, 4.34, 4.35.

1.0 Introduction

This paper has a twofold aim: an elucidation of the means by which a program is interpreted by another program, and the implications of this elucidation for software-system structure. It should be apparent that the mechanisms by which a program's semantic content is elaborated lie at the core of everything that takes place within a computer system. Therefore, it should not be surprising that an investigation of these mechanisms should yield a clearer insight into many different aspects of a software complex - compilers, loaders, file systems, encapsulated (i.e. virtual in the sense of [14,17]) machines - and their interrelationships. As for the mechanisms themselves, they lie at the edge of our general understanding of program semantics, and no claim is made for having said the last word on this subject. Indeed, this paper should be regarded as a stepping stone, in the sense that the control structure and terminology introduced later, while thought to be basically correct, represent a new way of looking at systems, and as such are not yet fully developed.

This paper grew out of an investigation of multi-programmed emulation, i.e. the co-existence of multiple microprograms in a mutually supportive environment. However, progress can be made on this problem only after it is realized (1) that the fact that an interpreter program residing in a special store or executing directly on the host machine is a special case of a more general problem: (2) that there are in general *layers* of interpretation in execution at any given time. It is this "wheels within wheels" nature of the problem which makes it confusing, exacerbated by the more or less general lack of structure in the situations where it occurs.

Indeed, it must be stressed that the discussion of interpretation mechanisms cannot be held in isolation from the environment in which they exist, i.e. the "operating system", because this is itself involved in the interpretation hierarchy.

1.1 Problem Statement

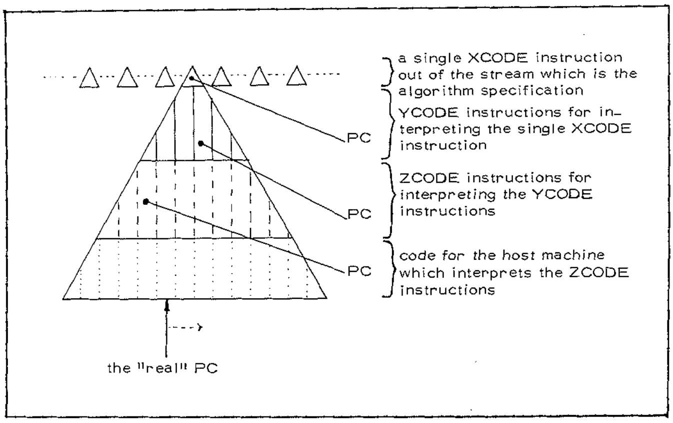
Although a great deal will be said along the way about compilers, loaders, and operating systems, the actual problem at hand is the derivation of a control structure for layers, hierarchies, or *nests* (as we shall refer to them) of interpreters. The term *interpreter* is a shortened form for "interpreter program", which is a program which defines a sequential *machine* as we commonly know them. Such a machine is called a *virtual* machine (in contrast to *encapsulated* machines, as mentioned earlier). There is a unique machine, called the *host* machine which lies at the bottom of any interpreter nest and which is not defined by any interpreter. This is another way of saying that the host machine exists outside of the control structure which encloses all other machines (interpreters). An *emulator* is thus an interpreter which is written in the code of, and executed directly by, the host machine. Emulators often are kept in a special store called *control-* or micro-store and are therefore often referred to as microprograms; these distinctions are however irrelevant to our discussion, and the more neutral term *hostmachine code* will be used instead, when referring to code at this level (or more generally, on the level of the *host* machine itself).

In order to help the reader to understand the generality which is desired of the final control structure, the discussion will refer to the following sample problems in interpretation:

P0. An interpreter may be written in any 'machine' code or language.

P1. A piece of microcode may be invoked as a procedure like any other procedure. Although there must necessarily be detailed differences, since a different store is being referenced, this problem ensures that the final control structure will not presume that such code (and host machine code in general) is qualitatively different from other codes in the system.

P2. Consider a procedure FRED written in XCODE i.e. running on the X-machine. This machine might be written in host machine code, but it might also be written in YCODE (which might be written in ZCODE etc.). This implies that the execution of FRED implies the nested execution of an arbitrary number of interpreters (including none) stacked up underneath it. Problem: run FRED on various XCODE machines (having different nests under them) without changing FRED or those (running on possibly different machines again) who call FRED.



a single XCODE instruction out of the stream which is the algorithm specification
YCODE instructions for interpreting the single XCODE instruction
ZCODE instructions for interpreting the YCODE instructions
code for the host machine which interprets the ZCODE instructions
PC
PC
PC
the "real" PC

Figure 1 An Interpretation Nest

For each XCODE instruction executed, increasing more are executed at each descending level. Although there are four PC's, only the lowest one has a true hardware interpretation.

P3. Expand an existing interpreter by adding *only* new operations, and later shrink it back to its original contents, all at run-time. This problem is a reflection of the data and operator definition facilities found in languages such as [7, 8, 9]: upon e.g. block entry, new semantics become available which disappear upon block exit. If one has a machine for such a language, it is reasonable to expand and contract this machine to reflect the elaboration of the program.

P4. Write interpreters which require e.g. garbage collection, all of which can share the same garbage collector. This means that interpreters can themselves call procedures which possibly run on different machines.

P5. Write an interpreter for a CDC 6400 plus N ppu's [12]. This implies that interpreters can be multiprogrammed and can have both shared and private store.

P6. Determine when the environment pointers (**ep**'s) of nested interpreters are or are not successive subsets of each other.

It bears mentioning at this point the more global problem which prompted the original investigation: in a system which allows user microprogramming, it should not be necessary to recreate software which already exists on other emulators. This implies that procedures which run on different machines can invoke each other, hopefully in ignorance of their "incompatibility". Besides the theoretical interest of the problem, there is a substantial economic one in avoiding reconstructions of device handlers, file systems, editors, and all the other forms of support software grouped under the rubric "operating systems". This means in particular that one could now construct "language machines" which only need take heed of the language's requirements, and not in addition I/O, synchronization, and other functions which are system, and not usually language, facilities.

1.2 Interpreter Nests

Figure 1 illustrates more exactly what is meant by an interpreter nest. An algorithm (or procedure or program, which terms will be used interchangeably hereafter) is viewed as a sequential stream of instructions. Each individual instruction is executed in its entirety by the interpreter (which defines the machine for which these instructions are intended). In general, the interpreter program itself executes a number of its own instructions whose cumulative effect usually includes a change in state of the program's data space (we confine ourselves, though without loss of generality, to non-selfmodifying code), and a change in the interpreter's state (e.g.

program counter (PC) update). It is worth noting that the interpreter has no further knowledge or interest in what the ultimate effect will be of the instruction sequence it is interpreting. Thus, the interpreter could be interpreting another interpreter, and itself be being interpreted.

If we look a little more closely, it appears that each instruction of the algorithm initiates, at each successive level of the nest, a series of procedure calls, i.e. control is from the 'top' downwards. That this is in fact not true will now be demonstrated. Consider the situation when an interpreter nest is to be 'deadstarted'. Execution cannot immediately begin with the first instruction in the algorithm on the 'top', since the interpreter is not yet necessarily initialized i.e. the interpreter itself must in general execute some set-up code which will define its initial conditions. This initialization process is necessary at each level of the nest, until the host machine is reached. Since the host machine is already initialized and running (indeed, it is the one who executed the code which triggered the deadstart of the nest), it is the only machine which can begin immediate execution of code. The execution of this code, which is the bottom interpreter in the nest, eventually reaches the point where this bottom interpreter is initialized. It then begins interpretation of code at the next level up etc. until finally the topmost interpreter is initialized and begins interpreting the first instruction of the actual algorithm. Thus it can be seen that the host machine is the primus motor, and that control emanates upward from the bottom of the nest.

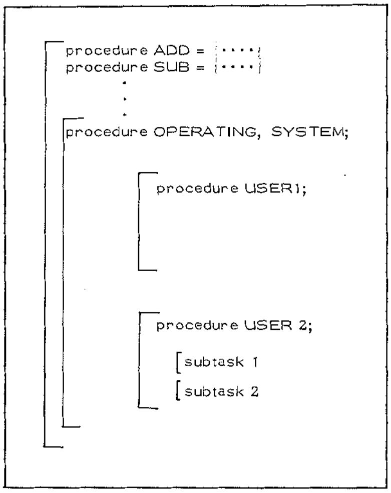


Figure 2 A Block-Structured View of a Typical Computer System

2.0 Well-Nested Interpretation Systems

In this section, we will attempt to fit interpreter nests into a very highly structured environment - that of Algol-60. This attempt ultimately fails due to its lack of generality, but it is a profitable failure nevertheless. We begin by first sketching the general environment into which the nests will be inserted.

2.1 The Basic System Model

The analogy drawn between machine instructions and procedures first hinted at in [16] is often dismissed as an interesting curiosity, but inspired by [1,15] and taken together with problem P1, we arrive at the preliminary control structure: the extension of the display to include procedures running directly on the host machine. For those not familiar with the Burroughs architectures and thus the implications of this extension, we now present a very brief outline which presumes a knowledge of how Algol-60 is implemented using a stack and a display; if this is insufficient, the reader is referred to [1,3].

The pervading philosophy of the Burroughs architectures is that the entire contents of main store, including the operating system, can be viewed as one huge Algol program. In the single task (mono-programming) case, the operating system is the global block and the user program a nested block. Clearly, this nesting can be represented in the display, which in fact does exist in the hardware, along with the display update mechanisms. If multiple users (multi-programming) are admitted, then each user is viewed (in analogy with the previous case) as a nested block withinthe global block, all these user blocks existing as blocks at the same (parallel) lexical level. The execution environment of each user (= task) is thus his display plus program counter. This arrangement, besides automatically separating the name spaces of the various users, allows the operating system to be invoked via the usual procedure call mechanism. In addition, each user when initiated is allocated a stack which will contain all of his activation records and local storage. Thus all code is reentrant, and e.g. an operating system call is built upon the user's stack. The utilization of Algol's control structures for a multi-programming computer architecture surprises many, but the result is in fact an elegant, general, and efficient model for what happens in a multiprogrammed computer system.

With the extension of the display, we arrive at the picture of a computer system shown in Figure 2 The machine instructions are viewed as the most global procedures in the system, and can be invoked by any program. Such a system would

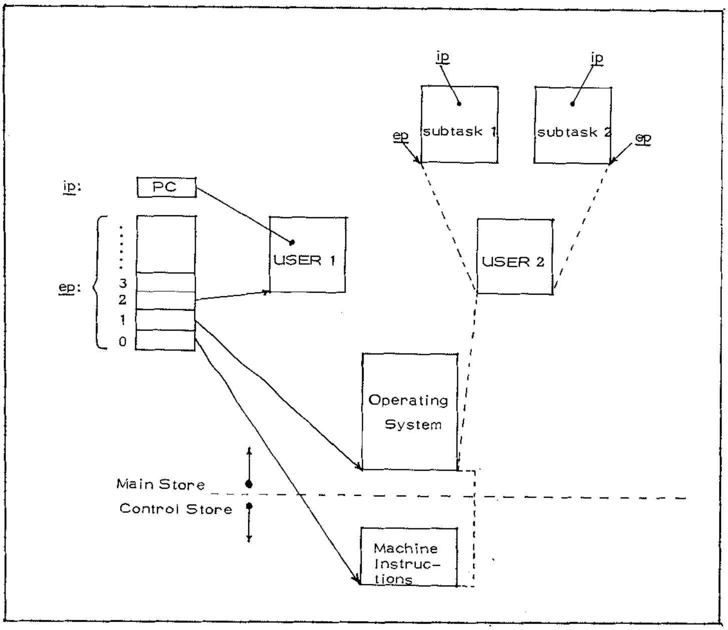


Figure 3 Stack and ep Structure of Figure 2-1

The dotted lines represent static (lexicographic) chains.

(after Figure 2) have the six stack areas as shown in Figure 3 , one for each of the tasks in the system, [although the Machine Instruction, System, and User2 stacks are dormant i.e. the activation records which they contain are currently passive as regards execution]. Using Johnston's terminology [2], the state of a task (process) can be represented by the couple (**ep**,**ip**), standing respectively for **e**nvironment **p**ointer and **in**struction **p**ointer. Since there are three potentially active tasks in this system (User1, Subtask1, Subtask2), each is represented as having an **ep** (its static chain) and an **ip** (its current point of execution). If we further assume that there is only one physical processor in the system, then the **ep** of (e.g.) User1 can be abstracted into the hardware display, and its **ip** into the PC of the processor.

Figures 2 and 3 yield several observations:

1. Invocation of micro-code as a procedure which is not necessarily a machine instruction (problem P1) is nothing out of the ordinary because the expanded **ep** now establishes a sufficient addressing environment to include this possibility.

2. There is nothing in the structure which requires only one layer of machine instructions i.e. an operating system which runs directly on the host (i.e. hardware) machine. This situation is found in all micro-programmed cpu's, and also in OS6 [18], which runs on an O-code machine (the target machine of BCPL [19], which itself runs on the host Modular One machine.

3. This structure helps greatly in the solution of problem P5 by supporting the necessary parallelism and data sharing. The interpreter aspects are of course untouched as yet.

2.2 The Boss-Machine Model

The system model just presented presumes that all the code in it runs on the same machine. This machine is that which is defined by the set of "instruction" procedures in the outermost block. The object code in this system can be viewed as a string of [lexical level, displacement] couples [ll,d] which reference the instruction procedures. It is important to distinguish among a reference to a procedure, specification of a call on a procedure, and the actual call of the procedure. The object code referred to is the first of these i.e. simply [ll,d] and nothing more. In particular, it is *not* the second (or by implication, the third) of these, since this is in contradiction to the principle of "bottom up" control mentioned in Section 1.2.

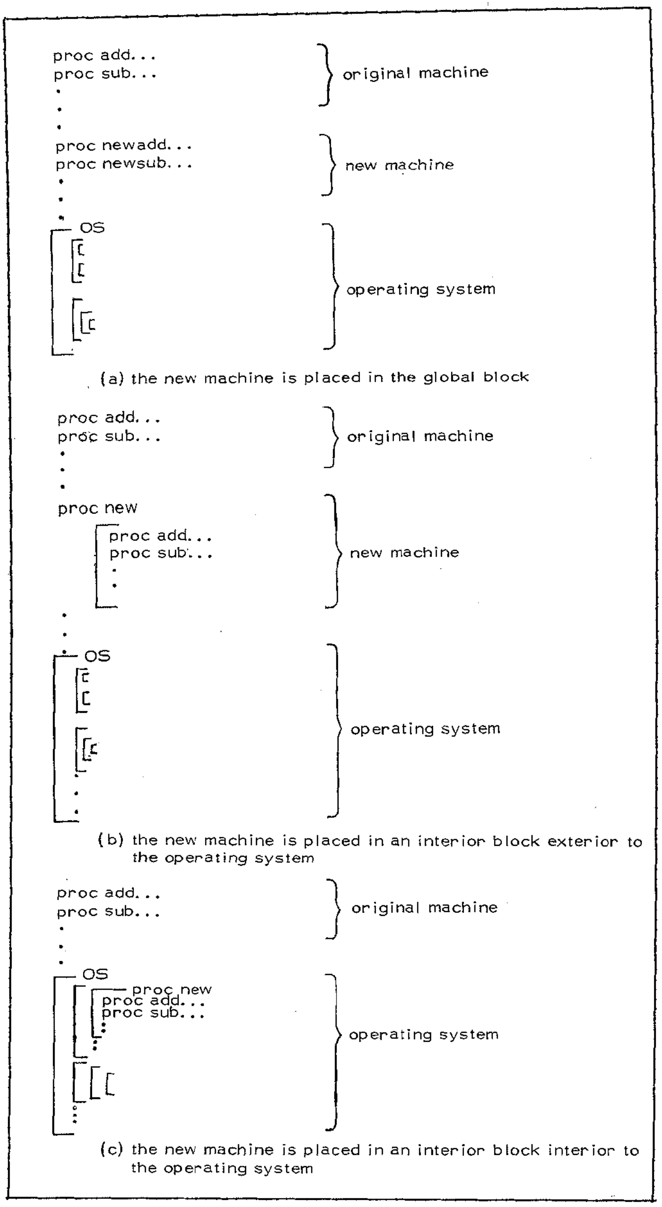


Figure 4 Possible Placements of a New Machine in an Algol-60 System Model

Returning to our object string of [ll,d] couples, we can see that it is the job of the host machine to sequentially fetch up an [ll,d] and perform a (block-structured) *enter* on the referenced procedure.

Recalling that our basic model at this point is an Algol-60 machine whose entire "contents" is one large Algol-60 program, we now proceed to include some new or foreign machines. There are essentially two possibilities: include the new operations in the global block, or include them in some interior block. The latter can itself take two forms: exterior or interior to the operating system block. Figure 4 illustrates.

The first possibility (Figure 4a) has to its advantage being a simple extension of the instructionset of the original machine, and as such is guaranteed not to introduce any new worries. If one is interested in merely expanding the given instruction set, this is the obvious strategy. On the other hand, if one is interested in introducing an entirely new machine, the fact that both machines reside at the same (and global) lexical level means that only language convention separates the two.

Possibilities (b) and (c) provide a lexical enclosure for the new machine via the procedure mechanism. Their differing placements, however, are critical. The new machine of (b) has the advantage of being visible to all other entities in the system; the price for this visibility is that its own environment consists only of the original machine. The environment can be expanded by invoking it from within the operating system block, passing procedures by name. By this means, access to e.g. operating system services can be provided. However, there is no way to avoid the fact that at least part of the machine must be written in the code of the original machine.

The new machine of (c) has no such problems - its environment automatically includes what (b) lacks, but at the converse price: lack of global visibility. This means that the only programs that can run on the new machine of (c) are those for whom this machine is lexically visible.Of course, this lexical visibility can, as with (b), be enhanced by an appropriate sequence of procedure calls with procedures as name parameters. Unfortunately, this approach becomes very unsatisfactory, both for (b) and (c), if additional new machines are nested within them, etc. The reader is encouraged to satisfy himself on this point.

In spite of these problems, which arise directly out of Algol-60's scoping rules, this model is nevertheless quite usable in specific situations e.g. where the host machine is very fast and one expects new machines only at this level (i.e. Figure 4b). Its lack of generality, which has a direct connection to problem P6, is in fact its greatest advantage from an implementation point of view.

we have referred to the "original" machine a number of times in our discussion. This was partly to avoid confusion with the host machine, on which the "original" machine runs. It is however more important to realize the role which this "original" machine plays - its instructions are always globally visible, and it is also the machine on which the operating system (i.e. resource allocation) executes. Because newly introduced machines necessarily play a subordinate role with respect to this "original" machine, we refer to it as the "boss" machine, and a multi-interpreter system as described here, which is structured after the precepts of Algol-60, as a Boss-machine Model of nested interpretation.

2.4 Secret Knowledge and the Boss-Machine System Model

The activity of a compiler in the Boss-machine model is the generation of a string of instructions from the input text, but because of the **ep**'s expansion to include the machine instructions as procedures, the generated code can be viewed as a string, not of integer opcodes, but of the **ep** addresses [ll,d] of the instruction procedures. The job of the host machine is to perform an *enter* in sequence upon each of the procedure addresses in the string.

Comparing this interpretation with contemporary reality, one can see two differences: (1) in reality, the opcode procedures are not globally visible, and (2) as a consequence the requested invocation must be represented by an ordinal (the integer opcode) rather than an **ep** address. The question we now ask is: how does the real compiler know which integers are associated with which instructions? In the first (idealized) instance, there is of course no problem since the associations are given in the compiler's **ep** and thus are statically known. Because in reality the instructions are not a part of the **ep** there is only one answer to the question - that the compiler obtained the necessary information from outside its **ep**. Since by definition the **ep** incompasses all of a program's 'knowledge', information such as the opcode-integer mapping is made available to the compiler as 'secret knowledge' by formally unknown means.

```
procedure ADD = {...}
procedure SUB = {...}
procedure Interpreter X =
        procedure ADD = {...}
        procedure SUB = {...}
        .
        .
        .
        procedure Interpreter Z =
            [
            ]
.
.
.
procedure Interpreter Y =
        procedure ADD = {...}
        procedure SUB = {...}
        .
        .
        .
        compiler to Y-machine
        .
        .
        .
.
.
.
    Boss machine operating system
            .
            .
            .
            compiler to Boss machine code
            .
            .
            compiler to X-machine code (XCODE)
        .
        .
        .
```

Figure 5 A Specific Case in Boss-machine Model

The deeper implications of secret knowledge are treated in Section 4, but it may be helpful to point out a few other examples of the use of secret knowledge commonly found in computing systems. An obvious example is Fortran's COMMON (both blank and labelled), which relies on secret knowledge supplied by the programmer to make the proper data associations. The same is true of BCPL's global vector, which in addition functions as the linkage area for separately compiled procedures, another aspect of secret knowledge. This linking problem is partially non-existent on a B6700 since external (library) routines are a part of the compiler's **ep** and hence have known addresses; other separately compiled procedures were historically disallowed, and allowing them clearly requires the use of secret knowledge to link them in. Another aspect of secret knowledge, which is left unpursued in this paper, is its relationship to program structure and maintainability as discussed e.g. in [21].

We have thus far constituted that in a system which includes the machine instructions as part of the **ep** of all programs running on that machine, the object code may be viewed as a string of **ep** addresses which can be generated without the use of secret knowledge. We now ask what happens if a compiler running in this system is to generate object code for a different machine? There are two possibilities: incorporate some secret knowledge into the compiler or include the instructions for the foreign machine in the compiler's **ep**. Since the former is a well known technique, we now explore the implications of the latter.

Figure 5 illustrates a particular instance of the inclusion of foreign or new machines into the Boss-machine system. With respect to the compilers in this system model, it is clear that the compiler for the X-machine must utilize secret knowledge, whereas the compiler for the Y-machine need not. In either case, however, the object code streams of these subsidiary machines may contain invocations of boss machine instructions (which might e.g. be I/O instructions). Thus the role of the boss machine is to establish a sort of default or communal operational structure which the subsidiary machines may obtionally choose to avail themselves of. They are forced to use this structure only when they desire to communicate with other parts of the system, since this is the only 'standard' available. This comment applies particularly to the invocation of other interpreters.

Of particular interest among the boss machine's instruction procedures are those called 'interpreter' which are the means by which the subsidiary machine's instruction sets are entered into the global **ep**. This particular instruction is distinguished from operations such as Add by having a semantically non-trivial internal structure, because in order to effect this structure it invokes the *enter* function of the host machine. An important omission in the illustrated Interpreter instructions is the list of parameters required for them to carry out their function. Most obvious is the need for an indication of what object stream the interpreter is to interpret. In the figure, the X-machine can be supplied with any X-code object stream, whereas at least in some instances, the Y-machine would expect to receive a copy of its compiler. The next missing parameter is a procedure (by name) from within the boss machine's operating system, which will be the means by which the emulator can invoke operating system functions (which are otherwise lexically invisible). This parameter is not essential except insofar as it is desired to avoid duplication of function, but is probably unavoidable if (as is likely) it is desired to share the file and mass storage subsystem, support for which is presumably available only on the Boss machine.

There remains, finally, the question of exactly why the Boss-machine model fails to satisfy our requirements of generality, as explicitly expressed in problems P2 and P6. One possible answer is that it is purely the fault of the scoping rules which we imposed, and there is a great deal of truth in this. Lying beneath this answer, i.e. the reason behind the reason, is that the lexical structure of a program, and the lexical structure of the programs which interpret the program are not necessarily the same. Just as scope and extent are coincident in Algol-60, so are the **ep** and **ip** structures of the Boss-machine model. The explication of these structures and the demonstration that this analogy is not accidental is the topic of the next section.

3.0 Interpreter Structure

The purpose of this section is to give more precision to the description of an interpreter program's structure and its relationship to other programs in the system.

3.1 The Nature of Interpretation

Up to this point we have assumed an intuitive understanding of what an interpreter program is. The fact that the problems P0-P6 exist at all suggests that interpreter programs possess some special properties which are not commonly found in other types of programs.

Consider the following list of properties of interpreters:

a. Most programs execute (never mind how) and deliver a result of some sort, whereas an interpreter program *by itself* is meaningless: it must have another program on which it is to operate.

b. The "data" for an interpreter program is a machine code i.e. data which inherently contains a special type of semantic information.

c. This semantic information is decoded by using a special construct called a program counter (PC) which contains, it could be said, the entire semantic future of the "data"s interpretation.

d. In general, interpreter programs can form nests, which implies that the regime of control is "bottom up" rather than "top down" as in a nest of procedure calls.

Property (a) is the weakest, since it is not difficult to find examples of programs which are not interpreters which possess the same property (data base managers, sort/merge routines). Properties (b) and (c) are closely related, both inherently containing the idea of a "position" in the data and its step-wise "development". This position and the extent of the development define what is called the data's state. We can now ask if there exist programs which are not interpreters which also possess this property?

Consider a sorting program. One could argue that the sorting key corresponds to a

```
STARTUP: PC := 0;
              .
              .
              .
              .
IFETCH:      MAR := PC;
             PC := PC + 1;

             goto OPTABLE (opfield(MDR));
             OPTABLE = (ADD, SUB, •••);

             ADD: •••
                  •••
                  goto IFETCH;

             SUB: •••
                  •••
                  goto IFETCH
             .
             .
             .
             .
```

Figure 6 Classical Interpreter Structure

```
procedure ADD = (•••);
procedure SUB = (•••);
      .
      .
      .
procedure array OPTABLE = (ADD, SUB, •••);

STARTUP:  PC := 0;
              .
              .
              .
IFETCH:     do forever
                    MAR := PC;
                    PC := PC + 1;
                    Readmemory;
                    OPTABLE (OPFIELD(MDR));
            end do;
```

Figure 7 Improved Interpreter Structure

PC, and the state of the sort process to the sorting "machine"'s state. [A data base manager can be viewed as a slow-moving sort in this respect.] A counter argument is that whereas the sort-key exists explicitly within the data set's space, a PC does not i.e. it is an abstraction which, while essential, exists external to the data. However, this is not getting at the heart of the matter.

One usually conceives of a program as consisting of code, and data which this code manipulates. Thus from the point of view of the interpreter program, the data of the program it is interpreting is at an additional level of indirectness; any changes made to this data occur as side-effects of the data (i.e. program) it is interpreting. In the case of a sorting program as "interpreter", the "code" it is interpreting is the data to be sorted, and there exists no further level of data wherein changes are made as a result of this "code"'s interpretation.

Therefore one can see that programs such as sorts and data base managers do not possess properties (b) and (c); it is possible that there exist other types of programs which do, and yet are not interpreters, but we have been unable to produce any. The same applies to property (d), which appears to be the strongest. Therefore, it seems reasonable to conclude (until a counter example can be presented) that the idea of a program counter and two data spaces (the code, and the data it changes) are characteristic of interpreters alone.

We now proceed to examine the internal structure of a typical interpreter with the hope that this examination will lead to a better isolation of those features which are unique.

Figure 6 illustrates the classical form of an interpreter. Due to its lack of structure (from the point of view of the structured programming debates [20]) it is not surprising that this picture does not yield much insight. A first step toward improvement can be taken by recalling the earlier postulation of viewing the individual instructions as procedures. With the exception of some messiness involving PC update in transfer-of-control instructions [perhaps in itself a veiled commentary on the "traditional" structure of same], Figure 7 can be viewed as an improvement.

The critical reader might point out that Figure 7 looks a little naked, since one usually expects to see such a collection of program statements surrounded by some

```
procedure ADD = (···);
procedure SUB = (···);
      .
      .
      .
procedure array OPTABLE (ADD, SUB, ···);
procedure Singlestep (PC, OPTABLE) =
      {bitstring PC; procedure array OPTABLE;
       begin
           MAR := PC;
           Readmemory;
           Singlestep := OPTABLE (Opfield(MDR))
       end;}

interpreter X (Singlestep, PC, OPTABLE) =
      {procedure Singlestep; bitstring PC; procedure array OPTABLE;
       begin
           STARTUP : PC := 0;
                        .
                        .
                        .
           IFETCH :   do forever Singlestep (PC, OPTABLE);
       end;}
```

Figure 8 General Interpreter Structure

sort of brackets. In the Boss-machine model, procedure brackets were introduced for this purpose, but given that model's lack of generality, it is important to be carefulon this point. The issue at hand is to arrive at a specification of what the activation record of an interpreter looks like. Having decided that interpreters are probably something special, there is no reason to suppose that a standard procedure activation record is sufficient. On the other hand, when viewed merely as a program, large parts of an interpreter are indeed ordinary procedures. Thus there are grounds to expect that the activation record for an interpreter will resemble that of Algol-60, with additions to account for its special nature.

It is possible to improve further on Figure 7 by reducing the Ifetch loop as shown in Figure 8. Two major changes have been introduced - a procedure called Singlestep and a new type of bracket "interpreter" (reflecting the fact we expect a different type of activation record). The former is introduced to emphasize the fact that the actual interpretation process can proceed as a series of procedure calls, whereas the actual startup process involves the creation of a special type of activation record. That the difference can be isolated to the start-up of the interpreter is consonant with the earlier discussion of the bottom-up control regime (property (d)).

3.2 The (ep,ip) Structure of Nests

In the preceding section, we discussed the structure of interpreters by examining an interpreter in isolation. We now broaden our horizons by including the program it is interpreting and the program which (in general) is interpreting it.

The critical insight is to realize that if **(ep,ip)** represents the state of some hypothetical algorithm running on some hypothetical machine, then **(ep,(ep,ip))** represents the state of that same algorithm running on an interpreter for that machine which is running on some other hypothetical machine. Stated in another way, the first **ep** is that of the algorithm, the second **ep** is the **ep** of the interpreter, and the **ip** is the locus of execution within the interpreter as maintained by the interpreter's machine. The **ip** of the algorithm exists somewhere in the **ep** of its interpreter and therefore does not appear explicitly. Thus in generalizing to a situation involving several interpreters in nested execution, the **(ep,ip)** couple becomes **(ep,(ep,(ep,(...(ep,ip))...)**. Note that the **ep** nesting sequence always ends in '**(ep,ip)**' which is to say that no matter how many interpreters one piles up, down at the bottom, churning away, is a *real* machine executing *real* instructions whose address is given by a *real* PC which is the **ip**. Note also that this **ip** is the only **ip** which occurs in the expression, which is another way of saying that only one machine is *really* executing, and all the other 'machines' piled on top of it are 'executing' only by virtue of the happy coincidence that their program code happens to represent an interpreter.

Let us now exactly define part of the (**ep**,**ip**) structure of Figure 5. It is clear that the **ep**'s for the X- and Y-machines are nested within the Boss's. This however implies nothing about the **ip** structure, which can be quite disjoint from the **ep**'s structure. For example, X-code could be written in either host-machine code (as is Boss) or in Boss code: which is the case cannot be extracted from the figure. In the former case, the (**ep**,**ip**) of a program SAM running on the Y-machine is

[A] $(\mathbf{ep}_{sam}, \mathbf{ip}_{x}) \Rightarrow (\mathbf{ep}_{sam}, (\mathbf{ep}_{x}, \mathbf{ip}_{host}))$

whereas the latter case is

[B] $(\mathbf{ep}_{sam}, \mathbf{ip}_{x}) \Rightarrow (\mathbf{ep}_{sam}, (\mathbf{ep}_{x}, \mathbf{ip}_{boss}))$

$\Rightarrow (\mathbf{ep}_{sam}, (\mathbf{ep}_{x}, (\mathbf{ep}_{boss}, \mathbf{ip}_{host})))$

Note that in either case the ultimate **ip** is that of the host machine, in agreement with the preceding paragraph.

3.3 God-given Operations and Memory

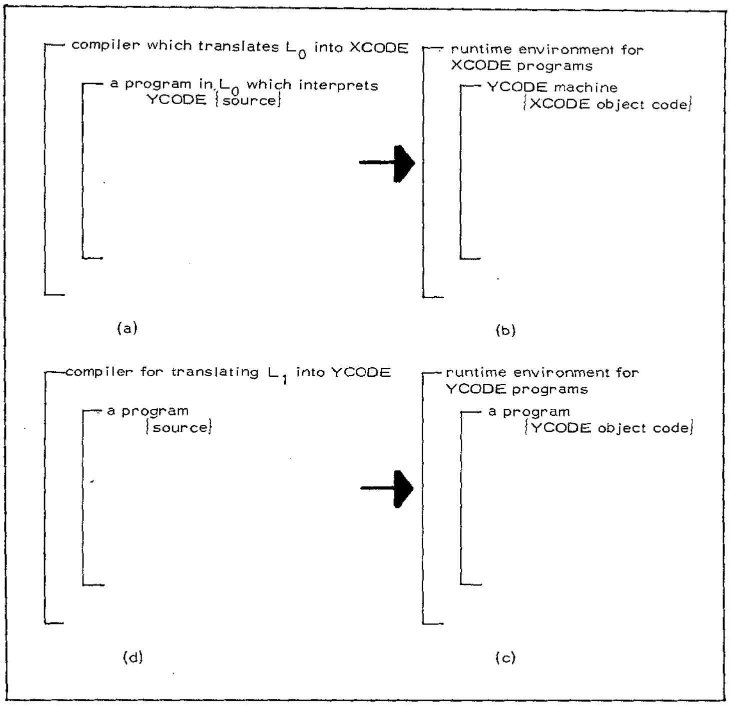
Up to this point in our discussion, we have assumed the presence of certain crucial operations, of which block (**ep**,**ip**) entry is the most obvious, but which also includes block exit and perhaps inter-task synchronization primitives. One might first be inclined to say that these operations should be defined as procedures fully analogous to e.g. the opcode procedures, but the following example illustrates that this cannot be the case.

Consider an Algol program running on a pure Algol machine, about to perform a block entry. Is it possible to write this operation as an Algol procedure, thereby logically obviating its need to be supplied as a hardware function? Clearly one runs into a recursive loop, since one must *enter* this procedure so it can perform the 'enter' operation. A little thought should convince the reader that this reasoning applies even if one postulates an Algol dialect which allows full access to the Algol-machine's registers and instructions (except *enter*). The lesson to be learned here is that *enter* is an operation which is presumed by Algol i.e. it operates outside (indeed it must) the confines of Algol's semantic space. Thus, as far as Algol is concerned, *enter* is a 'god-given' operation.

Although not usually recognized as such, god-given operations occur quite frequently in computing systems, examples being user supervisor calls in a machine having user and supervisor hardware states, and coercions in a typed language.

While it is true that the case of *enter* is particularly clear, the same reasoning applies to the hardwired operations of the host machine. Thus, the pure block structure model of Section 2 with its scopes expanded (ad infinitum) to encompass opcodes as procedures is ultimately bounded in its global extent by the wired-in instructions of the host machine. The various *enter*'s, *exit*'s, and hard instructions all (must) exhibit the same property: they may not appear as a part of an activation record in the space on which they operate. This does not, however, necessarily mean that e.g. *enter* must be a hardwired operation (although this might be a reasonable criterion to place on a new machine design) - it only means that such operations should be treated as ones which lie outside of the control structure herein described. As such, these operations can be executed by any machine code which executes outside the semantic space on which they are to operate; in our case, this machine code is the host machine but in an encapsulated machine [14,17], it would be the 'real' machine.

On the basis of the foregoing discussion, it is now easy to categorize the host machine's registers: they are god-given memory i.e. memory which exists outside the scope of the memory which it references; it is here that the final **ip** resides.



compiler which translates L_0 into XCODE
a program in L_0 which interprets YCODE {source}
runtime environment for XCODE programs
YCODE machine {XCODE object code}
(a)
(b)
compiler for translating L_1 into YCODE
a program {source}
runtime environment for YCODE programs
a program {YCODE object code}
(d)
(c)

Figure 9 The Various *ep*'s of Compile-time and Runtime

Items (a), (b), (c) correspond to the text; (d) is included for the sake of completeness. In general, each of the *ep*'s is disjoint from the others.

4.0 Non-Well-Nested Interpretation Systems

Section 2 discussed well-nested interpretation systems and introduced the Boss-machine model and the concept of secret knowledge. Section 3 treated the structure of an isolated interpreter, and that of its environment, and introduced the concepts of god-given operations and memory. In this section, we will weave all these threads together, and from the resulting fabric see the necessary structure for generalized nests of interpreters.

4.1 Compile-Time and Run-Time

In Section 2, we were able to ignore the distinction between information available at compile-time and that available at run-time i.e. the distinction between the nascent program's **ep**'s in these two phases of its life. This was possible because in a system with no secret knowledge, these two **ep**'s are necessarily identical. (The reader is encouraged to convince himself of this.) For the same reason, we were able to ignore the goings-on between the point when the compiler was finished compiling the program and the program executed its first instruction.

If we relax the ban on secret knowledge, these two simplifications are in general no longer valid, and we are forced to consider three environments where before there was only one. More precisely, these environments are (a) the compiler's (**ep**,**ip**), (b) the running program's (**ep**,**ip**), and therefore (c) the (**ep**,**ip**) of the machine which interprets this program. In the No secret knowledge case, the first of these was sufficient for all three purposes; in particular, (c) was the same for both the compiler and the executing compiled program cum run-time environment. See Figure 9.

4.2 Interpreter Nest ep and ip Structure

The implication of Figure 9 is that, as mentioned in passing in Section 3.2, the **ep** of the running program need have nothing in common with the **ep** of its interpreters. Furthermore, since the interpreter itself is a program, this statement is recursively applicable.

Thus in a nest of interpreters, there can be N disjoint **ep**'s, plus the **ep** of the program which is executing at the top of the nest. However, the disjointness of the interpreter **ep**'s does not belie the vertical nesting of the bottom-up control regime. Hence, there are two distinct structures required to describe an interpreter nest,

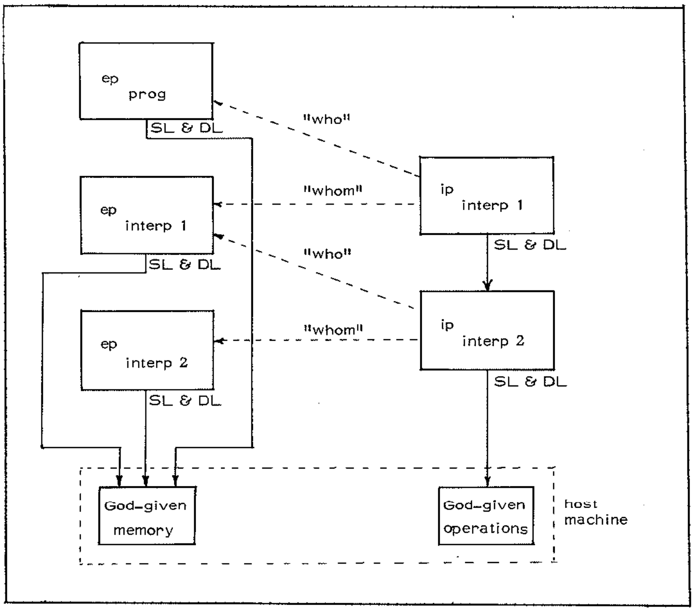


Figure 10 Interpreter Nest *ep* and *ip* Structures

one for the **ep**'s, and one for the **ip**'s. As regards these two sructures, the **ep** structure is that which contains *all* forms of data: constants, numeric values, addresses, and labels; the **ip** structure contains the relationship between the various nested interpreters: who is running on whom. The relationship *between* the two structures is found by answering the question of who "who" and "whom" are in the preceding sentence: "who" is the interpreted program i.e. its **ep**, and "whom" is the interpreter program i.e. its **ep**. Figure 10 illustrates.

The **ep**'s and **ip**'s shown in the figure should be interpreted as consisting of single activation records, for the sake of simplicity. Thus the static and dynamic links are identical in all cases. Clearly, no problem is introduced if each **ep** consists of multiple activation records - the static and dynamic linking of them is as usual. In the case of the **ip** activation records, within a given nest, the static and dynamic links are identical, but when a procedure call is made which runs on a different nest, then the structure of the links is as shown in Figure 11. It should now be apparent how procedure call-by-name would operate: the actual procedure parameter must consist of an **(ep,ip)**, which if different nests are involved, consists of a pointer to the topmost activation record in the **ep** of the procedure and a pointer to the topmost activation record in the **ip** of the procedure, plus a PC value for the topmost program. [Theoretically, a PC is necessary for each of the **ip** activation records, but this is unnecessary if the convention is established that no interpreter state is saved which is not on an Ifetch boundary.]

We have thus far shown, in a constructive manner, that an **ip** activation must consist of a pointer to the **ep** of the interpreter, a pointer to the **ep** of the interpreted program, and static and dynamic links to earlier activation records. In the interests of problem P3, it seems desirable as well to include OPTABLE in the **ip** activation record, but we are unable to produce a more compelling argument than this. Figure 12 shows the final form of an **ip** activation record.

4.3 The Generalized Emulator System Model

Let us now reconsider the Boss machine model of Figure 5. The procedures Interpreter X, Interpreter Y, and Interpreter Z may now be placed anywhere within the structure, since the relaxation of the secret knowledge ban allows the **ep**'s of the interpreter compiler, the executable program, and interpreting machine to be disjoint. A further implication is that the instruction procedures of the Boss machine itself need no longer be global either. Thus we can return to a system model whose lexical structure superficially resembles the B6700, with the operating system most

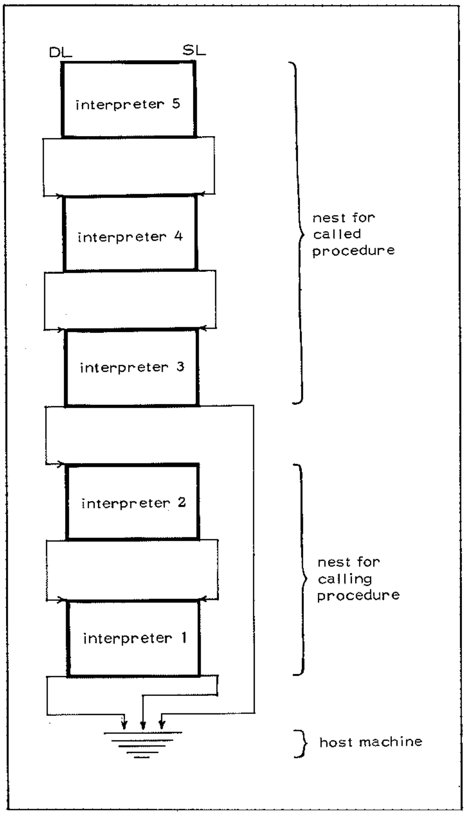


Figure 11 Static and Dynamic ip links

global, and other entities located arbitrarily within its inner scoping structure.

We now place items (b), (c), and (d) of Figure 9 at arbitrary points within this structure. If the placement is such that the compiler of the program (d) can find the interpreter (b) in its **ep**, then the activation of the program can be arranged via a call to the interpreter with the compiled procedure as a parameter. Notice however that this will not work if the run-time environments of the program and interpreter are non-null and disjoint from the compiler's **ep**.

Furthermore, if we assume that the **ep**'s of eventual interpreter nests are disjoint, as in general they will be, then the same problem arises. What is needed is a means to access a program (which might or might not be an interpreter) which is not lexically visible. This can only be accomplished by assigning the entity, along with its environment, to a global variable. This is to say that the principal requirement of a general multi-emulator system (particularly one which supports nesting) is a retention discipline [2,4,5,6]. Indeed, a general form of retention is required, one which allows the assignment of functions.

It is interesting to note that the retention requirement falls directly out of the relaxation of the ban on secret knowledge, which itself is related to the definitions of scope and extent. While it is perhaps premature to draw a direct causal relationship, it should nevertheless be apparent that there is at least an indirect one. In either case, it is worth examining how contemporary systems which allow secret knowledge manage to either avoid or hide the required retention mechanisms.

In today's systems, the most comprehensive source of retentive information is the file system, and especially various sub-program libraries. These, combined with the crucial services of a linkage editor, provide the bridge from the compile-time **ep** to the (potentially dual) **ep** of run-time. In light of this discussion, it should be no surprise to recall that systems with no secret knowledge need no linkage-editor i.e. the retention function which is supplied by a linkage editor is unnecessary in a system which, by its nature, generates no need for a retention discipline.

If, on the other hand, attention is directed to the loading function (as distinct from linkage editing), the common practice of using the opportunity to achieve initialization of variables e.g. Fortran's COMMON, can also be seen as an example

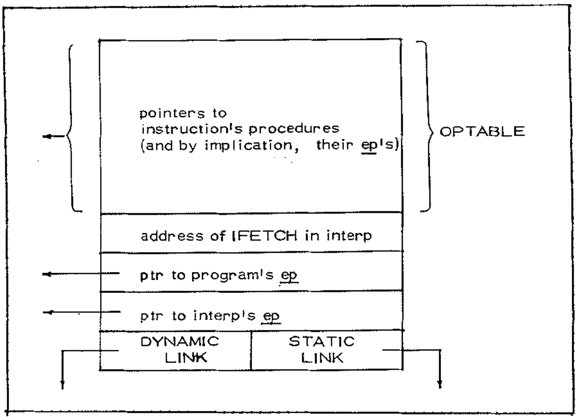


Figure 12 ip Activation Record Fine Structure of a hidden retention discipline. For this reasoning to apply in the strict sense, however, it should be postulated that the e.g. Fortran program be an interpreter program whose presence is invoked by the loading of a program which it is to interpret. On a slightly different tack, Algol-60's problems with initialization of *own* variables can be seen to stem from it's lack of a retention discipline.

5.0 Observations and Conclusions

A model has been presented which supports nests of interpreters in a general manner. The intentionally theoretical discussion has precluded a number of observations of a more pracical nature, e.g.:

1. Most systems will have a particular machine on which the "operating system" runs i.e. all other machines are started up after this one. This first machine is started up by performing the (god-given) *enter* function on it. If the machine is *enter*ed again, the result will be to create an encapsulated version of the same machine, thus illustrating how the model can reflect the reality of systems such as [17,14].

2. While we have introduced the requirement of a retention model for storage management, we have not placed any requirements on it besides the ability to assign procedures. Thus, the models of [4,5,6] are all sufficient for the implementation of a general interpreter nest system. The latter has in addition the ability to model LISP-like languages in addition to Algol-like ones, thus making the primitive functions suggested there in good candidates for the corresponding god-given operations for the host machine.

3. We have said nothing about how one communicates between the **ep**'s of programs running on different nests. Clearly, there is a problem of data conversion and standards to be solved if cross-machine calls are to become a viable tool. As regards the addresses themselves, if we logically admit that an interpreter invocation deserves a full address space in which to work (i.e. a 0 to MAX addressing space), then the concepts of interpreter activation and segmentation can be seen to have a very close relationship. Indeed, one can easily imagine an entire system built on this principle, thereby uniting compilation, job startup, execution, procedure linkage, and main and background storage allocation, all under the same regime.

4. It should be recalled that the retention requirement grew out of the need to make environments of otherwise disjoint interpreters available to each other. This need stems primarily from the nesting of interpreters, but also from problem P2 and the desire to make software running in one environment generally available to all. In systems where e.g. all interpreters run on the host machine [10,11,13 and many more], the retention requirement is

relaxed, since the environment to be retained is trivial. Thus, all that remains is to model the **ip** stack.

5. The **ep** and **ip** structures which have been presented are the means by which a collection of interpreters and their program libraries can be knitted together. It is important, however, to realize that these structures place no restrictions on the way an individual interpreter is written, i.e. they come into play only when an interpreter is started or ended, and when a cross-interpreter call is made.

The title of this section also promises some conclusions, but our feeling is that the most reasonable thing to conclude is that we have barely scratched the surface of the question of what is actually happening when a program is interpreted. We, just as the programming language semanticists, have found the sequential nature of program execution to be a most deverly tied (Gordian?) knot. We hope we have added a useful new blade to computing community's Swiss Army knife.

Acknowledgements

I would like to acknowledge Nigel Derrett's sine qua non midwifery of these ideas.

References

[1] Organick, E.I. Computer System Organization. The B5700/B6700 Series. Academic Press 1973.

[2] Johnston, J.B. "The Contour Model of Block Structured Processes". in ACM Sigplan Symp. on Data Structures in Programming Languages. Feb. 1971.

[3] Randell,B. and Russell, L.J. Algol-60 Implementation. Academic Press, 1964.

[4] Chirica, L.M. et al. "Two Parallel Euler Runtime Models". ACM/Sigplan/Sigarch/IEEE Symposium on Higher-level Language MachineArchitecture. Nov. 1973.

[5] Berry, D.M. et al. "On the Time Required for Retention". ibid.

[6] Bobrow, D.G. and Wegbreit, B. "A Model and Stack Implementation of Multiple Environments". CACM 16, 10 Oct. 1973.

[7] Griswold, R. et al. The Snobol 4 Programming Language. Prentice-Hall, 1968.

[8] van Wijngaarden, A. et al. Revised Report on the Algorithmic Language Algol-68. Dept. of Computer Science, University of Alberta; Edmonton, Alberta, Canada. March 1974.

[9] Lindsay, C.H. and van der Meulen, S.G. Informal Introduction to Algol-68. North-Holland Publishing Co. 1971.

[10] Wilner, W.T. "B1700 Memory Utilization". FJCC, 1972.

[11] Bell,G.C. and Newell, A. "The IBM System/360 - A Series of Planned Machines Which Span a Wide Performance Range." Computer Structures - Readings and Examples. McGraw-Hill, 1971.

[12] Thornton, J.E. "Parallel Operation in the Control Data 6600." FJCC, 1964.

[13] IBM System/370 - Principles of Operation. Form No. GA22-7000, IBM Corp., Data Processing Division, White Plains, N.Y.

[14] VM/370 System Programmers Guide. Form No. GC20-1807-3 for Release

2 PLC 13. ibid.

[15] Hauck, E.A. and Dent, B.A. "The Burroughs B6500/B7500 Stack Mechanism". SJCC, 1968.

[16] Wilkes, M.V. and Stringer, J.B. Microprogramming and the design of control circuits in an electronic digital computer. In Bell, G.C. and Newell, A. Computer Structures: Readings and Examples. McGraw-Hill 1971.

[17] CP-67/CMS IBM doc. No. 360D-05.2.005 Rev. 6/69. Program Information Dept., 40 Saw Mill River Rd., Hawthorne, N.Y. 10532, USA.

[18] Stoy, J. and Strachey, C. OS6 - An Operating System for a Small Computer. Computer Journal 15, 2 and 15, 3 1974.

[19] Richards, M. BCPL Reference Manual. Cambridge University, Computing Laboratory, 1969.

[20] Gries, D. On Structured Programming. CACM 17, 11 Nov. 1974. (letter)

[21] Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules". CACM 15, 12 Dec. 1972.