"END;" "FOR <for-list>;" "DO <statement>" and so on. If we also demand that the scope of certain language constructs such as block structure be indicated by the numbering of the program as outlined above, we achieve three things:

(1) with little trouble we can alter a parser to handle incremental segments and link them together correctly [Li 70, Wu 70];

(2) correct writing of programs, with respect to program structure, is encouraged by this method; and

(3) with very little work beyond that needed for (1) above, a prompting parser can be constructed which prompts the user for the next allowable segment, whenever its structure is unique and can be determined by the statement numbering and the state of the parser.

In section 3B3 we will define a small language which will be used for examples in the thesis; and we will define the set of allowable non-terminals as segments for that grammar, and give examples of its use.

Wulf [Wu 70] has described a method for cementing together incomplete program segments, using the numbering rules and segmentation just given. Basically, the parser is given a set of non-terminals which may be used as segments. Whenever a statement is parsed which requires but does not have such segments, they are left "dangling" in the parse tree and are threaded on a list in an order inverse to the numbers of the lines from which they dangle. If a line is entered which matches that segment, then they are hooked together, provided that the line numbering for the two segments is consistent. This last requirement simplifies the parser problem, since the line number (or its lexical position-plus-indentation) uniquely specifies the set of dangling non-terminals in the thread with which it can be mated: no match means there is a temporary syntactic error (it may not necessarily be permanent), which is just another form of dangling non-terminal which may later be connected correctly.

Deleting a segment simply places the non-terminal on which it hung back on the thread of dangling/suspended non-terminals. All of

is made to perform a useful service.

```
1   BEGIN
    1A   FUNCTION B;
         1A1   IF N ≠ 0
               1A1A   THEN B ← F
               1A1B   ELSE B ← 0
    1B   END
    1C   FUNCTION F;
         1C1   N ← N - 1;
         1C2   F ← A + B * A;
    1D   END
    1E   N ← 1;
    1F   A ← 1;
    1G     F;
2   END
```

Figure 3B2-1: Hierarchical Statement Numbering

If we call each number or character in a label a "segment," then the depth of a statement in the lexical hierarchy is exactly equal to the number of segments in its label. Hence, 1A1A is at level 4 in the hierarchy; its parent is 1A1, and its successor at the same level is 1A1B. Later we will show how we propose to connect the labelling of statements with the ability of a parser to handle incomplete programs. It must be stated that such numbering, while necessary, would not necessarily have to be shown constantly to the user; indeed, correct indentation is probably more useful to him, and if used consistently could obviate the need for BEGIN and END in Algol for instance.

Recent research into incremental parsing [Li 70] has shown that allowing segments of text to be incrementally parsed with no constraints on the type of allowable segments can be very expensive. Wulf [Wu 70] suggests the use of some specified set of non-terminals of a language as allowable segments. Such a set for Algol might contain statement segments such as "IF<exp>," "THEN <statement>," "BEGIN,"

3B2 Adaptation of Tree-Meta for an IPS

Among the properties often stated as desirable for the writing of programs in an IPS are the following:

(1) the ability to enter the text of a program in any order, and to delete, insert, or change statements of the program without having to restart execution of the program from scratch;

(2) the ability to interlace program execution and modification of the program as an aid to debugging;

(3) the program, being subject to change must also be subject to review; thus the user must be able to display the existing text of his programs;

(4) the program is amenable to change and editing within the system itself; put another way, the program is data of the language.

This list is not exhaustive, but it supplies enough constraints to allow us to attack the problem of adapting a Tree-Meta parser to the requirements of an interactive system. Behind the above properties lie some more fundamental facilities which programs and programmers of an IPS need.

If text is to be changeable, then we must have a means of talking about a particular statement. The ability to point with a light pen on a CRT display, say, does not completely fill this requirement since programs must also be able to reference elements (it is, after all, programs which will be manipulating them, even in the light pen situation). Thus, some scheme of numbering or indexing is necessary. Since programming languages often have a hierarchical lexical organization - such as the block and compound statements of Algol - and since we intend to map programs into a tree structure, we will use a tree labelling device similar to Dewey decimal, which device was developed by Engelbart et al [EER 68] and has proven useful. Figure 3B2-1, a simple program, is an example of this labelling scheme. Notice that, by interspersing numeric and alphabetic characters in a label, no decimals are needed and the space normally taken by them

they are separated by a "/" in Tree-Meta instead of the "|" used in BNF. The "$" means "zero or more occurrences of the following"; and .ID, .NUM, and .EMPTY represent pre-defined syntactic entities (identifiers, numbers, and the nil element).

Using this grammar, the parse tree corresponding to the expression A+B*A is

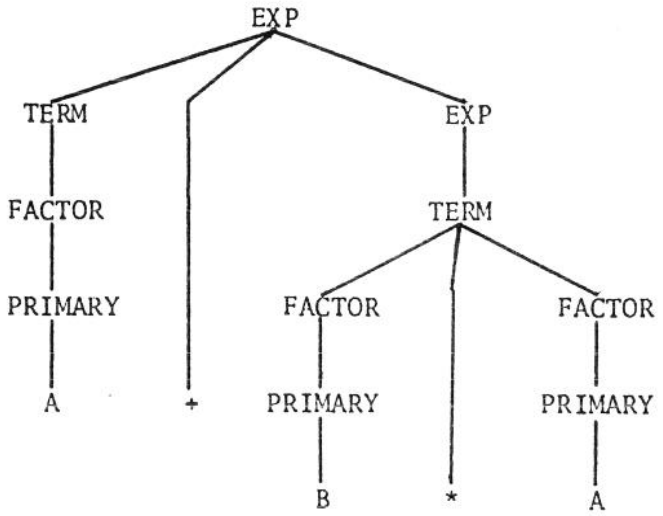


Figure 3B1-3: Parse Tree for A + B * A

This form of parse tree contains much more information than is needed for our purposes, and Tree-Meta allows one to create a parse tree such as the following, from the same grammar (with some added frills):

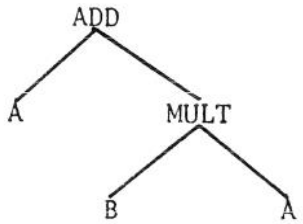


Figure 3B1-4: Simplified Parse Tree for A + B * A

The reader is referred to [EER 68] for a description of the mechanism for accomplishing this transformation. For our purposes, we will use this operator-oriented tree representation since it is readily adaptable to the use to which we wish to put it, namely executing programs.

3B Using a Parse Tree to Drive an Interpreter

We have seen in section 3A a number of different methods of interpretation, at different levels: source, lexically scanned, and parsed representations of statements. Now, we will speak of interpreting parsed statements, but instead of representing the parse in postfix notation, we will use a parse tree. Parse trees provide a fairly well understood means of viewing the structure of statements in a context-free language.

3B1 Tree-Meta Parser

Parse trees may be produced and represented in a number of ways. The Brooker-Morris system [BM 62] and more recently a meta-compiler system called Tree-Meta [EER 68] have used parse trees as an intermediate representation of a parsed program for compiling purposes. We propose to use parse trees in an IPS to provide the ability to interpret and compile programs within one system. First, however, we will give a very brief outline of the Tree-Meta parser and its extension to an interactive environment.

The following BNF [Na 63] describe a small expression syntax in much the same class as that given in the Algol 60 report.

```
<exp> ::= <term> + <exp> | <term> - <exp> | <term>
<term> ::= <factor> | <factor> * <term> | <factor> / <term>
<factor> ::= <primary> | - <primary>
<primary> ::= <identifier> | <number> | ( <exp> )
```

Figure 3B1-1: BNF for Small Expressions

This can be written in the Tree-Meta format as

```
EXP = TERM ( "+" EXP / "-" EXP / .EMPTY );
TERM = FACTOR $( "*" FACTOR / "/" FACTOR );
FACTOR = "-" FACTOR / PRIMARY ;
PRIMARY = .ID / .NUM / "(" EXP ")";
```

Figure 3B1-2: Syntax for Small Expressions in Tree-Meta

In this notation, parentheses have been used to group alternatives;

3A2B Polish Postfix as Used in LC^2

In the APL system, relatively little time is spent parsing statements as compared with executing actions. This is due primarily to (1) the emphasis on array-oriented operations, and (2) the simple expression syntax of APL. The LC^2 syntax [VZ 69] is more complex than APL's and it is not a language oriented toward array operations. Thus, LC^2 is interpreted at a lower level than that of APL, namely, at a point after the level of parsing. LC^2 statements are parsed as they are entered into the system. The result of this parsing is a string of interpretive code which is essentially a form of Polish postfix notation [RR 64]; this form of code is well suited to a stack discipline for its execution. For example, the Polish postfix (hereafter called simply "postfix") for the expression A+B*A is A B A * + which is read from left to right and can be interpreted as the following sequence of operations:

(1) push the value of A onto the stack;

(2) push the value of B onto the stack;

(3) push the value of A onto the stack;

(4) multiply the top two items on the stack together and replace them by the value (B*A);

(5) add the top two items on the stack, thus producing A+(B*A), and replace them by the result.

Thus, the fetch part of the interpreter paradigm for postfix code is a very simple left to right scan. The variables are actually represented by an address - the symbol table address - in LC^2 and each operation, including the placing of the value of a variable on the stack, is represented by a code number. That code is then used to get the address of the appropriate routine from a vector of addresses. The F-level of the paradigm is thus very simple and efficient. The main cost of interpretation in LC^2 is therefore due to the X-level of the interpretation: the checking of variable types, checking for undefined variables, handling subroutine calls, etc. The main costs in JOSS, by way of comparison, are in symbol table searching and parsing; and in APL the overhead of interpretation is found mainly in walking around transition diagrams and doing type checking on variables.

of the interpreter which recognizes basic operands as sketched above. Above each arc in the diagram is the "name" of some lexical unit of the intermediate code of APL\360 statements, and below each arc, enclosed in a square, is the name of the routine to be executed if that arc is traversed by the interpreter. The control state of the interpreter is defined by a pointer to some node in a transition diagram, along with a pointer (or program counter) to the next lexical unit in the statement being executed. The routine to be executed next is determined by picking the arc emanating from the current node whose associated lexical unit matches the next lexical unit in the statement. If such an arc is found, the interpreter control is updated to point to the node at the other end of that arc, and the routine named below that arc is executed. If no arc label matches the next lexical unit in the statement, a syntax error is indicated. Where there is only one arc between two nodes (such as the one between $\alpha$and β in figure 3A2A-1), the name of a lexical unit may be replaced by the name or label of a node in a transition diagram (such as EXPRESSION on the α-β arc). The interpreter will then call itself recursively and interpret from the node named. In the above diagram for instance, BASIC is called from another diagram containing a node labelled EXPRESSION and EXPRESSION may be called by BASIC when a parenthesized expression is encountered.

> There are thus two controlling devices for the interpreter:
> (1) the intermediate form of the statement being executed, which is scanned lexical unit by lexical unit from right to left and which is used to control movement in the transition diagrams, and
>
> (2) the transition diagrams, which can direct the order in which actions are performed, and which are actually a syntax recognizer for the APL language.

Also, there is another level of interpretation at the level of whole statements in order to accomplish the normal sequencing from statement to statement, as well as explicit transfers of control within the user's program.

put on an execution stack; results of operations are left on the same stack. Then, whenever an operator or function is encountered, its left-hand operand will be the next item immediately to its left in the pseudo-code string. If the item to the left is a right parenthesis, the current operation is suspended until a matching left parenthesis is reached during execution of the parenthesized expression. At that time the operator which was held in abeyance may execute, and be assured that its left operand is on the top of the stack, while its right operand is immediately below the left in the second stack position. Monadic (unary) and dyadic (binary) operators are distinguishable by checking if the item to the left of an operator is itself an operator or not.

In order to make this mode of interpretation possible, each APL statement which the user enters is divided into lexical units which are easily distinguishable by a numerical code placed at the right end of each unit. Thus, vectors are preceded at the right by an indication of the number of constants in the vector, and its type (integer, real, or character). Also, as mentioned above, identifiers are replaced in this internal statement representation by pointers to symbol table entries which in turn have pointers to the locations where the values of the respective variables are located.

The actual means of controlling the execution of an APL program is a device known as "Conway transition diagrams" [Co 63,BL 68]. A typical diagram from [BL 68] will demonstrate how the F-level of the APL interpreter works.

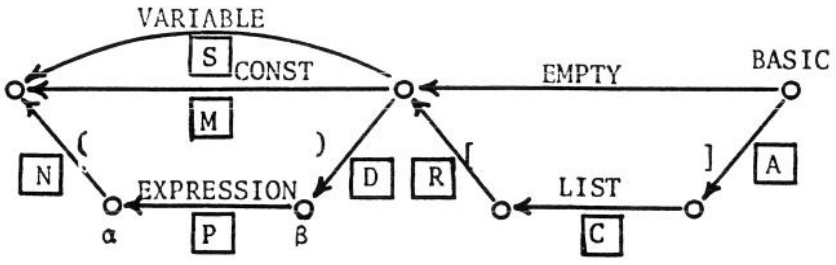


Figure 3A2A-1: Sample APL Transition Diagram

Figure 3A2A-1 is a simplified transition diagram for that part

It is inefficient because most operations involving variables or references to program statements require a table search in order to ascertain the location of that variable or statement, and because execution entails reparsing the statements each time.

3A2 Interpretation on a Pseudo-code

Since statements in a program are changed infrequently in comparison to the number of times they are executed, it seems reasonable to remove some run-time burden from the interpreter by translating the source text into an intermediate form for execution. By doing this, the operations can be indicated by a string of instructions of "pseudo-code" for which the F-level of the interpretation paradigm becomes simple relative to the manner in which a system such as JOSS accomplishes it. Furthermore, if we guarantee that the symbol table entry for a variable will never be moved (although its value might be), then this pseudo-code does not require a table search for every access to a variable, but only an indirect address operation via the symbol table entry since the code may contain symbol table addresses. Note that reparsing on each execution of a statement will still be necessary, however, since all that we have done is the equivalent of a lexical scan on the program text.

The APL\360 system [BL 68] is a good example of an interpreter which operates at this level.

3A2A APL Interpretation via Transition Graphs

In APL, operators do not have a precedence ordering as they do in Algol 60 and FORTRAN, for instance. Rather, the following rule is used:

> "every function takes as its right-hand argument the entire expression to its right, up to the right parenthesis of the pair that encloses it."*

An implementation of this rule simply involves scanning the pseudo-code for a statement from right to left. When a simple operand such as a variable name or a constant is seen in this scan. its value is

---

* Iverson, K. E. and Falkoff, A. D. "APL\360: User's Manual", IBM, Aug 1968: p. 3.3

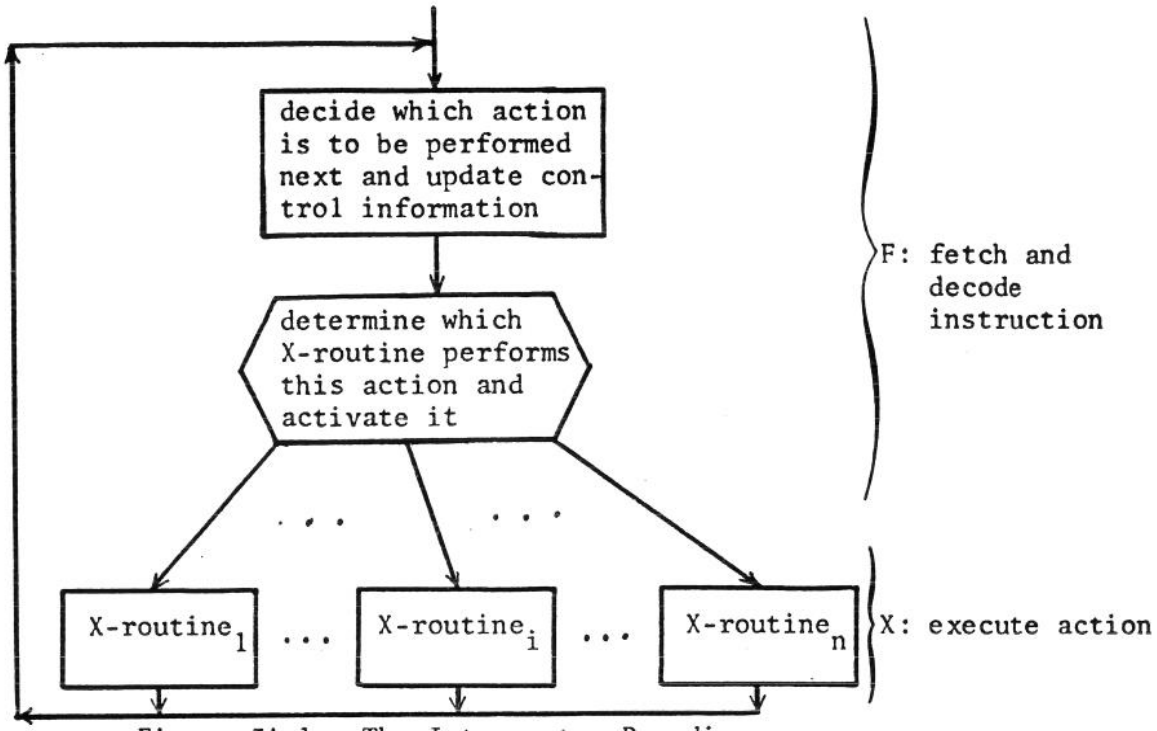


Figure 3A-1: The Interpreter Paradigm

the interpreted program, and these routines may be highly interconnected. Some of our results will deal with that set of routines and how they are structured, while others will be concerned with the "fetch" cycle of the pseudo-machine represented by section F of the diagram.

3A1 Full Interpretation from Source Text

Some interactive systems such as JOSS [Ba66] use the source text of a program as typed in by the user as the code for the interpreter. Then the action of proceeding from command to command can be fairly complicated since it involves parsing the input string in order to determine the command sequences. Also, since the X-level of the interpreter must associate the name of a variable with its value, a table lookup is required on every use of a variable. This form of interpretation is the most general and the most inefficient: it is at the flexibility-inefficiency extreme of the execution spectrum. It is the most general because changes to the program text by the user are possible and are automatically recognized by the system since it uses exactly the same representation for the program as the human.

below indicate before and after in a lexical sense; we will only use before and after to denote time relationships.

3A Simple Model of Interpretation as a Pseudo-Machine

In one sense, an interpreter for a language is necessitated because the potential complexity of machine code to execute programs correctly is very large when compared to the operations available on most computers. Interpreters bridge this gap by simulating the action of a pseudo-machine whose operations are at a level high enough to easily handle such run-time complexity, using a fairly simple "machine code." Figure 3A-1 is a model of a simple interpreter. The n routines at the X-level perform the actions of the pseudo-machine and the remainder of the paradigm reflects the machine-like control mechanisms for executing successive instructions.

Some actions such as loops and control jumps are performed by routines which modify certain information of the interpreter such as the program pointer. Also, most of the complexity of the system is in the X-routines since that is where the actions are performed on behalf of

3 Interpretation and Compilation in an IPS

Many of the features of an IPS initially appear to demand a large number of run-time decisions in order to execute programs correctly. Typeless variables, the detection of undefined variables and possibly incomplete programs are examples of such features. Systems in which much of the semantics of a program is decided at the time of execution are called interpretive systems. That is, they interpret the meaning of each action in the program when the time comes to execute it, unlike most traditional compiler systems in which the entire program is translated into machine code and then executed.

In this chapter we will make some initial sorties into the possibility of having a system which is interpretive but also able to, at least partially, compile code. That is, the system will act interpretively when necessary, but "produce" machine code for execution so long as the semantics of the program and its variables remain constant (with respect to their semantics -- not necessarily their values) over some period of time. We will refine these notions after an historical sketch of some models of interpreters.

Throughout the remainder of the thesis we will be speaking about objects which are related in a lexical manner (such as the text of statements in a program) and events related in time (such as the execution of statements one after another). In order to distinguish between these two cases we will adopt the convention that above and

Indeed, it is well known that two supposedly compatible Fortran compilers on at least one large manufacturer's line of computers do not execute all programs in the same way. Besides, the verifying procedure, no matter how informal, must be redone whenever either of the compiler or the interpreter is changed.

We propose instead to investigate the possibility of obtaining a means of producing a true interpreter which is also a compiler; i.e., the programs defining the compiler are an integral part of the interpreter and the same physical instructions are used when interpreting a given operation in a source program as when producing compiled code for that same program. The next chapter will begin this investigation.

(program p_x) = (a valid representation of algorithm x)

And, if the algorithm x is not well-defined, the problem is even more difficult. The analogy suggests the following: in the act of finding a concrete representation of an algorithm, the variables of that search, which are programs and the necessary data structures, become more and more constrained and less variable, the more closely they approach a valid representation of x. This means that even in an interpretive system requiring no declarations there will be a tendency for de facto bindings to appear as the program approaches correctness. That is, the program itself can be expected to change less and less, and the type and/or structure of variables will be changed less as the program is debugged. Indeed, such de facto bindings exist for relatively long periods of time very early in the creation/debugging process, when compared with execution times. A variable which initially is used as a number is not likely to vacillate rapidly between being a number and naming an entire array, for instance. The program-directed prompting for definition of program parts, described in the section Assumed Control in our evaluation of LC^2, causes the higher level control of a program to become somewhat stable very early in the development of a program.

Such de facto bindings can be a connection between the spectral extremities of interpretation and compilation. For, if we could take advantage of them and produce compiled code for executed program actions, that code could be expected to remain valid and useable for a reasonable time. The constraint on this, of course, is that we be able to revert to interpretation should that code ever become invalid due to a change in any of the bindings on which it depends and that the user need not be aware of this occurring.

What form should such an interpreter-cum-compiler system take? The first notion is to write a separate compiler and interpreter for the language with mechanisms to allow mixtures of interpreted and compiled programs to execute together. We claim that it is highly improbable that such an undertaking, for any reasonably large language, will be successful. It reduces to the problem of being able to prove, at least to one's self, that two large programs behave in the same manner on all inputs which each could accept.

2H Interpretation and Compilation

It has been mentioned that interactive languages must become efficient and powerful as well as flexible to use if they are to be of real value to those who use computers for large tasks - such as building operating systems. And the bootstrapping issue forces this even more. The problem which remains is how to get efficiency and flexibility, two ends of a common spectrum, to co-exist.

We first need to understand the extremes; the bulk of this thesis will then deal with resolving their differences. We define interpretation as a method for executing programs which determines the exact effect of each operation in the program by using the context available to it at the time that the operation is actually performed. By context we mean the data space defining the programs and their data, and the data objects used to communicate within the interpreter itself.

Classically, compilation can be described as the preparation of a program for executing on some machine wherein the specific effect of each operation in the program is determined prior to execution, once and for all, using only the context available at the time of compilation. These definitions will be made more precise in the next chapter.

Thus the context within which an interpreter operates is, in fact, two separate contexts, that of the interpreter and its control and semantic information about the program, and that of the program and its data structures and control. These two may interact, causing changes in each other; this mutual interaction can also be considered a distinguishing characteristic of interpretation since a running program which was compiled can clearly have no effect on its own compilation since that is already finished. Basically, then, an interpreter allows a program(mer) to be imprecise about certain bindings (on the type of variables, for instance) by using the bindings available at the times those objects are used.

The process of creating and debugging a program can be viewed as an iterative, root-finding procedure where the equation to be solved can be stated as

which is accessible in a consistent, simple (and minimal) way and which is useable by components of that same repository.

Given that he can also access portions of other people's "personal libraries" in as easy a fashion (by adding their name or identification to the qualified name of an object), the system then can become a medium for collaboration between him and his colleagues.

2G Monitoring

The notion of an interrupt has been a great aid in the evolution of operating systems, but ,except for the ON-statement of PL/I, has not appeared in higher level languages. Some of those interrupts, such as checking each use of a variable or a statement, took advantage of the structure of the language. Such a facility would be very useful in an IPS.

Fisher [Fi 70] has described an interesting control mechanism called "continuously evaluating expressions" which mimic a separate processor which evaluates some expression and interrupts the running program whenever the value of that expression changes to true. If this were imbedded in an ON-statement facility, anything having a name in the system could be monitored (although the expense might vary widely depending on the object being monitored).

Another approach to monitoring is achieved by constructing the system as a configuration of sequentially cooperating processes, as suggested by Krutar [Kr 69]. Then a monitoring process could be inserted between any two system processes and would become active whenever control came to it. At that time it could decide whether conditions warranted making some program active. Thus it acts like a polled interrupt, unlike the first method which, while it may be implemented by polling is nevertheless distributed in scope over the execution of the program which activated it.

systems, the main operations of copying a file between secondary and primary storage has been a special operation. If we regard files simply as nameable groupings of data structures (i.e., programs, data and control information) which happen to reside on secondary storage, they begin to look more like other data structures.

Assume that copy is a monadic operator which, when prefixed to the name of some object, suppresses any normal evaluation associated with it. Then,

A ← copy B

where B is a procedure and A the name of a file means that a copy of the procedure B, which might include its text and object code, for instance, is to become a nameable object in A on secondary storage. To name an object D residing in a file A we can use the notation A.D; thus,

C ← copy A.D

would mean that a copy of the object D in file A is to become the "value" of the variable C in the default "file" which is the user's working memory. Neither of these operations have depended on the fact that A really is a file; it could as well be a function with a local variable B, or a program containing a local function named B. Indeed, it should be possible to simply access an object D in A with the same notation: hence A.D[I], A.D+3, or A.D(3,) might all be valid uses of the object D in A, depending on its semantics.

In such a system where everything is accessible by some qualified name (or tree name), such naming should be all that is needed for accession, whether the object be a file or program or a scalar variable. The "special nature" of a file then melts away, and more importantly, the notion of an explicit directory associated with secondary storage disappears. The user can then simply think of a universe of objects which he may use in a continual way from day to day for his work. This does not imply that he does not realize that storage hierarchies exist which have differing response times; economics will generally make him careful about such things. But it does mean that he can think of such a system as a repository of personal knowledge, algorithms, bibliographies, research ideas, etc.

stratum, providing certain functions to the user, which is layered on a system which provides the capabilities for those functions to operate. This can be considered a limited form of bootstrapping, and one is then led to the obvious questions: how far down can that top layer be pushed; how much of an IPS is needed to provide a base on which to build a full IPS; if one could find such a base, what primitives would it have and how many; if the number of primitives is small, does that necessarily imply that one could move an IPS built on them from machine to machine with much less trouble than the entire system? Our aim in the next three chapters will be to attack these questions one by one.

One very large trouble spot still exists, however, if we would like to bootstrap an IPS. The better interactive systems have been interpretive, and interpreters usually run many times slower than compiled code for the same language (often in the range from 15 to 50 times as slow); then the parts of the system which were written in the language would run n^2 times as slow (where n is the slowdown caused by the interpreter) because they would be interpreting the actions of a program while themselves being interpreted. Clearly, then, a bootstrapping IPS will need to be efficient as well as flexible and powerful.

Another very beneficial side effect of bootstrapping an IPS is that the system implementors are able to reap some of the benefits of using an interactive system. Moreover, after the system is complete, many system "bugs" which appear can be hunted down and corrected using the IPS itself, and the error-checking facilities provided by it - assuming, of course, that it is not one of the error-checking routines which has the bug.

2F Other Features

"The ability to retain programs and data in a variety of states over an indefinite period of time" [MPV 68] has been stated as a feature desirable in an IPS. Each of BASIC, JOSS, APL, and LC^2 has had some facility for files of programs and/or data and/or control information, but none has had all. And, in each of these

the interface ports, which is desirable but not entirely necessary. We will assume that either is possible.

Krutar calls programs which communicate input and output over ports, independent processes, independent in the sense that they care not from whom they obtain input nor to whom they pass output. A group of processes hooked together by their ports is called a configuration, and in fact has all the characteristics that a single or atomic process does. Such a configuration is also called a society of "sequentially cooperating processes" (not to be confused with Dijkstra's "cooperating sequential processes" [Di 68]), since control can reside in one and only one of the atomic processes at a time, flowing from one to another, along with interface data or parameters, over port connections.

This control/interface regime is not limited to connections only at the ultimate I/O level, but pervades the entire construction (configuration might be a better term) of the system. Thus, in the above diagram, it may very well be that the process labelled IPS is not an atomic process, but itself a configuration.

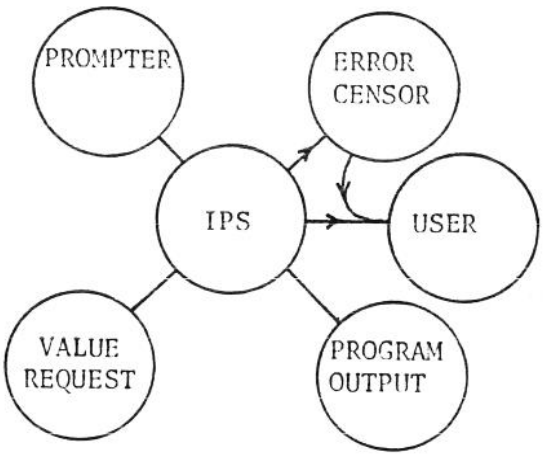
Systems constructed as process configurations have another property of interest to interactive programming. Since an individual process is not, in general, aware of which process is attached to a given port, one can insert a process between any two directly connected processes (and therefore an arbitrary configuration). For debugging this means that a user could interpose a process of his own into an IPS between any two system components which are processes. Good places for this might be between an actual input process and the translator - this would be a macro processor; or between an error routine and an output process - this would be a message filter.

2E Bootstrapping

Once a user's programs have access to some system data structures, many of the system functions can then be written in the interactive language itself, as just outlined. Thus there is a top

terminal. And if they are not knowledgeable about a given circumstance, then the immediate answer is that they must be augmented in order to handle that case. This kind of centralization will make USER and CENSOR unnecessarily complicated.

In any IPS there is (normally) a fixed number of routines which may potentially wish to interact with the user: the routines which print error messages, read lines typed by the user, or type program output are examples of these. Each of these routines can be said to have one or more "interface ports" to the user, which ports are unidirectional, two being needed if both input and output are to be performed. The generalizations of USER and CENSOR could then be a number of programs which can be attached to these ports (the same program may, of course, be attached to more than one such port). This distributes the interface and its control so that no one large routine is required to handle all cases; rather, each such interface is changeable (since the program attached to that port is replaceable) and may present a different form to the user depending on its function. A graphic view of such a system might be



where the ERROR CENSOR is connected, via its own interface port to the USER, with the understanding that USER will return control along the path which it obtained it; i.e., either to ERROR CENSOR or to IPS.

An excellent implementation of this model of programs with interface ports has been developed by Krutar [Kr 69] to whom this model is also due. That implementation uses coroutine control over

interrupt by the user. The IPS thus makes the tacit assumption that by calling USER it has fulfilled its responsibility for contacting the human user. This opens some interesting control possibilities since USER could in fact decide to handle some conditions without bothering his creator - interfaces for neophyte users of the system or control filters for an experienced user, designed by himself, are only two of the possibilities open.

But USER can only exercise such control if it has the information necessary to make decisions; that information clearly must also be available to its human counterpart if he is to be able to make good decisions. Hence, USER must have access to at least some of the control information of the system such as the call hierarchy, the control point in the user's running program, and so on. But USER is just a program in the language of the system and is therefore indistinguishable from any other program - except by its name, which does have special signifigance to a number of the system components. In the future, any arguments about the structure of an IPS dependent on the existence of USER must, therefore, be understood as applicable to the language as a whole.

USER can be viewed as an input filter to the IPS since all input normally passes through it. Its counterpart is a filter for the output from the IPS. We shall call it CENSOR, and the system knows about it just as it knows about USER and considers its responsibility for output to the user dispatched by calling CENSOR with the output as parameter. This does not mean that simple output capabilities do not exist in the system — they must if CENSOR is to be able to type anything at the user's terminal — but it does mean that the system relinquishes the action of typing on a console to one program over which the user has control.

2D Coroutine Control for Interaction

The foregoing requires that the procedures USER and CENSOR, having centralized control over I/O, must be "omniscient" about the meanings of all possible input and output strings at the user's