actions to be performed when such values are used;

(b) the number of operators in the BLM is very small (39 different instructions) because only a single instruction (such as ADD) is needed for all the data types — since each value specifies its own type.

Restricting the permissible operations on address values has two consequences:

(1) it is impossible to overwrite data in a structure (hereafter called a set) which a program does not have access to (legitimately, that is) because the limit of a set is contained in any address value referring to that set;

(2) the indirectness of some value to be used by a program can be transparent, since any commands can, using type information, discover such indirectness and automatically perform necessary indirect addressing; and

(3) an operation which would create an incorrect address is caught at that time instead of many instruction executions later, when such an invalid address can cause extremely obscure errors after possibly catastrophic damage to data in memory.

Protection is extended to programs since they cannot be overwritten. Also, because the exact bit structure of instructions need not be known, one more level of potential "bit twiddling" can be removed from programming.

The BLM does have some restrictions, however, as well as a few areas in which it could be profitably extended. For instance, since each value in the BLM has an extra word of information associated with

vector which serves as the memory of most computers. Indeed, one has not paid the price for this transformation by implementing the initial design, but must pay "rent" as it were, in terms of storage allocation and de-allocation programs to maintain those structures in memory.

The structures required in an IPS are, as can be seen from the preceding chapters, not trivial. And, since storage management is even more crucial in an environment where one of the functions is as knowledgeable and unpredictable as the human user, these problems are of primary concern.

Iliffe's Basic Language Machine (BLM) [I1 68] differs from most computers in several important aspects:

(1) memory has structure, and the overhead for representing that structure in a linearly addressed memory (including memory management) is a function of the BLM;

(2) addresses are not manipulable with the abandon allowed on most computers, but can be used by a command set sufficient for the purpose; memory protection is a very nice side effect of this, as it is on machines such as the Burroughs B-5500.

(3) programs are not allowed to treat themselves as data: programs may be treated as data only by a basic function of the BLM, its Assembler; normal programs can do nothing with programs but execute them.

The implications of these three points are impressive. Each basic item of the BLM has associated with it, type, size, and other information, and hence

(a) one can have representations for undefined values and allow

complexity is handled by the CBM. Moreover, such a feature is desirable even in a system which is not expected to be moved from machine to machine, because the notion of what should or should not be considered primitive by the CBM may depend to a large extent on measurements gathered about the performance of the IPS once it is in "production" use. Bottlenecks might then be eliminated by incorporating specific operations or data structures as primitives in the CBM, especially if the CBM were implemented as microcode on a microprogrammable computer.

In the remainder of this chapter we will further develop a set of characteristics for the CBM, without giving a specific instance from the set of machines which have those features. Appendix A contains a design for a specific CBM, and is a good example of a member of this set. However, it is not the only possible CBM, and there is nothing sacred about the design (except to this author, being the father); the design could be changed in a number of ways without being non-representative of the set of all CBMs.

5E1 Relation of the CBM to Iliffe's BLM

In building any large and complex system, no matter how well understood it may be, the process of creating and testing the components, or modules, still presents an enormous task. Iliffe [I1 68] has made the point that some of this complexity, rather than being inherent in the algorithms used, is caused by the present form of computers. One culprit to be singled out is the costly process of mapping the data structures natural to some process onto the linear storage

5E Design Considerations for a CBM

The CBM is intended as a base for an IPS, providing a set of appropriate primitives upon which the IPS may be built. Those primitives are suggested by the algorithms and data structures developed in the preceding chapters.

While we can generate characteristics which we would like those primitives to have, there still remains one major question: how much of the complexity of the IPS should be in the CBM, and how much in the rest of the system? One constraint on this is that the CBM should be kept as small as possible in order that it have some "inherent" exportability. On the other hand, operations which are common in the IPS, such as subroutine calls, ought to be primitive in the base.

Another consideration is that the CBM should "hide" the ideosyncracies of the hardware, such as different forms of arithmetic instructions depending not only on the type of operands, but also on whether one or both of them are in registers, etc. Hence, the CBM would probably have only one ADD instruction, and the CBM Factored Interpreter (abbreviated CBM/FI) would decide which particular hardware ADD to use in a given case. But this implies that the CBM knows the types of operands: it follows that the CBM data structures must contain information about their type, size, etc., as an explicit part of their "value," much like Iliffe's BLM.

Of course, any particular implementation of a CBM/FI will make specific design decisions as to how much is done in the "hardware" of the CBM and how much in the IPS. Hence, the IPS must be able to use the CBM is such a way that it depends only minimally on how much

Of course, they are much more since they are in a suspended control state and can be restarted from where they left off at some previous session. This is a very useful feature for humans who need to be able to set aside one task in order to work on another for a while, no matter for what reason. It is also useful for tasks whose lifetime is by nature very long,such as a personal budgetting or accounting system, for instance.

In the next section we will give a set of design considerations for a CBM, and we will be assuming a model of an IPS such as just given.

process connected to the other end of the port is activated. Later, when control flows back across that port, the suspended process becomes active and proceeds from the point at which it used the port. This is in contrast with subroutines whose control and variables become non-existent when they return control. Such coroutine control may be between two processes contained in a common process, between two processes having nothing in common save the user's GLOBAL process, or between processes belonging to different users. Indeed, logically, processes should be allowed to execute in parallel as well as sequentially, as in the RC-4000 Multiprogramming System,for instance [Ha 69].

Creation of a process from a prototype has much in common with a subroutine call in that parameters may be passed, space is allocated for the new process's variables, and control is passed to the process. The only difference is that new processes need to have unique names assigned to them, since their lifetimes are not as well ordered as are the incarnations of a recursive function, for instance.

The creator of a process can be treated much like a subroutine caller, by causing return-codes to be passed (along with control) to the creator whenever an abnormal situation occurs in a process created by it — except for the important distinction that the "returning" process is only suspended, and not necessarily completely deleted. Examples of such conditions are linkage faults (attempting to activate a port which isn't attached to any other process) or a willful request by a process to be deleted.

If processes can be kept around from interactive session to session, it is useful to think of them as "files" in the normal sense.

implementation of LC^2: each call had a set of return-codes which it could accept from a called subroutine. The zero return-code implied a normal return. If a non-zero return-code acompanied a return from a subroutine, two situations could occur:

(1) the return-code was in the set specified by that call, or

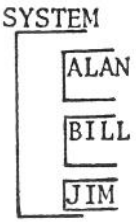
(2) it was not.

In case (1), each return-code specified by the caller had a label associated with it; on that return-code's appearing, control simply transferred to the appropriate labelled statement. In case (2), a return was done for the caller back to his caller, passing the return-code back. The only other addition to this scheme was the option of the caller to transfer to a specified statement whenever a non-zero return-code not anticipated by him appeared. This allowed certain routines to "clean up" and restore certain critical values to some state before allowing the return-code to continue propagating backward.

5D2B Coroutines

Coroutines were discussed in section 2D. There we described a model due to Krutar [Kr 69] in which instances of coroutines are created from prototype programs (looking much like normal subroutines) which communicated via "ports." The creation of a coroutine (or process—they are really the same concept) involves the establishment of a stack to contain local variables and control information within the process, since it may contain and use subroutines. When a process activates a port (by asking for input over that port, or attempting to move output across that port), it is suspended, and the

of ALAN, BILL, and JIM would have the following scope of names (and names only — this has nothing to do with control!) in such an IPS:



Exactly how objects are shared between processes belonging to different users with sufficient security guarantees is discussed by Lampson [La 69] and we will not delve into it here.

5D2 Control Within an IPS

Given a universe of discourse as we have suggested, what can one do with it in an IPS? Since the named elements of the universe are programs and values, it is clear that at least we expect to execute those programs, which will operate on the values. The point of concern is with the means of controlling execution. (An excellent source book for a treatment of control structures in general is Fisher's doctoral dissertation [Fi 70]). We will deal with two primary means, subroutine and coroutine control.

5D2A Subroutines

The only non-standard concept which we have advocated in connection with subroutine control might be termed return control. By this we mean the ability to place control at some point (in the past) in the call hierarchy, distinct from the normal one-level return from a subroutine. A simple mechanism for accomplishing this was used in the

may contain subprocesses which at their level are mutually independent. If A is a process, and B an object such as a variable or a function within that process, then its name, reflecting its heredity, is A.B (this was originally discussed in section 2F).

Not every process is contained in another super-process (or parent process), and so each user has one process which we will call GLOBAL, which contains all his other processes. Since containment by a process is a naming phenomenon, each process possesses a symbol table which "describes" all of its contained objects: a symbol table is part of the description of a process. Of course, not only processes may possess names; functions and Algol-type blocks may also have names which are local to them.

There is another level of naming which involves the IPS and its variables. Since we are advocating bootstrapping as a means of IPS contsruction, its variables must be named in the same manner as the user's. Moreover, since it is most often the case that an IPS resides within a time-sharing operating system, facilities are needed for users to share objects among themselves. For a single user, the GLOBAL process (and its Global Name Table, the GNT) establishes a means by which his processes may share variables. And, at the system level there is a process called SYSTEM which may be said to contain (a) each user's GLOBAL process, and (b) objects which are to be shared among several users (we have called these EXTERNAL). A better notation for GLOBAL would be a unique identifier for each user, such as his "account number" with the system. If we use first names of people as account numbers (for the sake of example), then a user population consisting

5D A Model of an IPS

A model of how an IPS should be structured has been dispersed throughout the foregoing chapters. We will here summarize that model. We do this for two reasons:

(a) to provide a unified summary of that model, and

(b) to supply structure for the discussion of features needed in a CBM (which discussion is begun in section 5E).

This model of our envisioned IPS has two main aspects of interest to a user of such a system: the naming of objects and the control structures which relate those objects together as programs and values.

5D1 The Named Universe of an IPS

People generally divide large tasks into smaller ones, grouping some tasks together because of some commonality of purpose (in their design), and making these groupings as independent of one another as possible. Such independence simply means that two such groups share (almost) nothing implicitly with one another. What they do share is generally expressed as a "language" called an interface (inter-faith?) language, and by defining these interfaces well, it is felt that one can gain some control over the complexity inherent in large systems [Di 69]. Also, in the day to day process of formulating and attacking problems, some task groupings are independent of others for the simple reason that they are not "solutions" to a common family of problems.

We have been calling such groupings processes, after Krutar [Kr 69]. More precisely, by a process is meant a collection of named objects whose identity is subordinate to the process's name. This implies a tree-structured naming convention for "objects," especially if processes

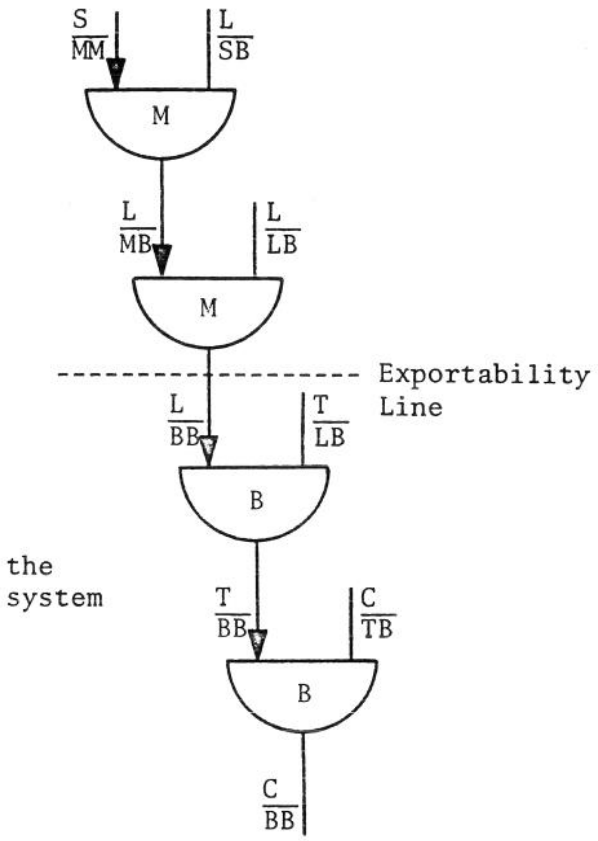


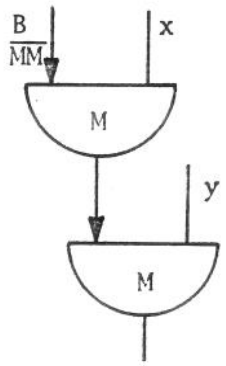
Figure 5C-2: Bootstrapping with a CBM

In the diagram, everything below the dashed line is independent of [M] and therefore easily exportable. Writing $\frac{L}{LB}$ now can be seen as a means of including L below that "exportability line" by making it independent of [M]. Indeed, the system can be moved between machines by rewriting the machine [B] for the new machine and simply moving everything below the exportability line onto the newly implemented CBM on the new machine.

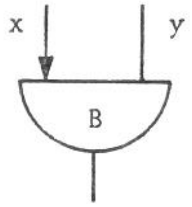
factored interpreter. So, while the IPS would undoubtedly itself be faster if written in S, say, it is nevertheless the case that the speed of programs in C is not dependent on this fact as it would be in a purely interpretive system where the efficiency of the underlying system is a large factor in the speed of execution of programs written in that language.

5C6 A Simplified Bootstrapping Model

We can simplify and clarify figure 5C-1 if we replace the parts of the diagram having the form:



by a transition of the form:



This transformation is analogous to viewing B as a machine in its own right. Then, recalling that we need $\frac{B}{MM}$, no matter how obtained, we can change figure 5C-1 to 5C-2 below, in which the bootstrapping procedure stands out more clearly.

T is not in any way dependent on the host computer [M], since it is written in L and produces parsers expressed in B.

5C4 The Initial IPS

Once T and L are available, the remainder of the system can be implemented using them, T for syntax and L for the more basic routines of the IPS. The system's growth beyond the primitive routines, such as the arithmetic routines, can be fairly amorphous: as soon as new C-constructs are implemented, they can be used in implementing other C-constructs. Indeed, some routines may be entirely written in a subset of C, without using L at all, or may contain mixtures of C, T and L. Step IV, therefore, is not as simple as figure 5C-1 would seem to imply, and realistically it should be viewed as a number of iterations over that section of the bootstrapping. During this phase, much of the system is self-implemented, thus giving its implementers many of the interactive facilities which the user of the finished system will have.

5C5 The Final IPS

It is worth repeating that the final version of the IPS (and, in fact, any version during the bootstrapping sequence) will run at the hardware speeds of the host computer since B is written as a

5C1 A Factored Interpreter for the CBM

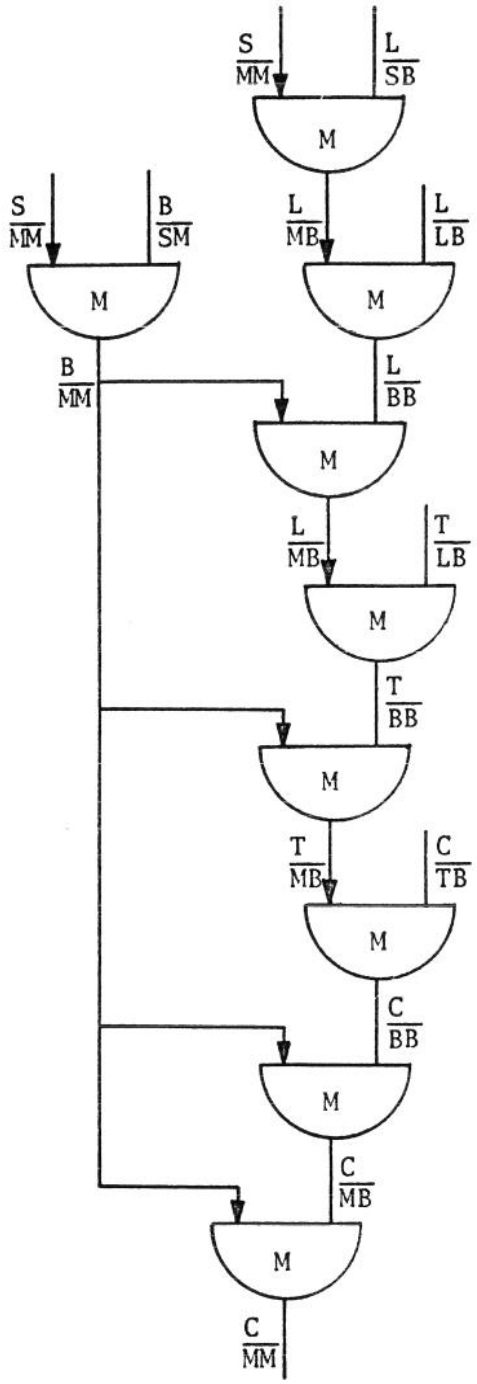
Since the CBM is the base upon which the IPS is to be built, it must be implemented before anything else. It can be written in any language S which is suitable and compilable on the machine[M]. We call the machine language of the CBM, B; it is written in the form of a Factored Interpreter in order that anything expressed in B will be able to run both interpretively and as compiled code for [M].

5C2 The Conversational Base Language

Once B is running on [M], we begin the task of producing L, the "assembler language" for the "machine" [B], to translate programs in L into programs in B. For the first implementation, L can also be written in the language S. The next part of this bootstrapping step, writing L in itself to produce object programs for [B], is done solely for ease of exportability of the system later. Once $\frac{L}{BB}$ has been obtained (the last part of Section II in figure 5C-1), moving L to a new machine [N] can be accomplished without reprogramming L at all; only B, the machine language of the CBM, written as a Factored Interpreter, needs to be moved by hand to new machines since L runs on [B] itself, which is invariant under such movement.

5C3 The Parsing Language

The parsing language T is similar to that found in chapter 3 (an adaptation of Tree-Meta [EER 68]) and is used in building the final conversational system, whose language is C. T is assumed to produce the type of parse trees needed by the TFI. Note, also, that



I: implement the CBM interpreter in S, running on [M]

II: implement CBL (=L)

III: implement the parser, T (T includes L as a sublanguage)

IV: implement the interactive system $\frac{C}{BB}$

V: use CBM FI to obtain final running system $\frac{C}{MM}$

Figure 5C-1: Bootstrapping Network for an IPS

Programmable extensions to the CBM data structures, flexible control mechanisms, and filing (and program saving/restoring) capabilities are among these items.

5C Bootstrapping an IPS using the CBM

Without going into the details of the Conversational Base Machine until later, we will outline a procedure for bootstrapping an IPS using it, and its language CBL (Conversational Base Language). An important feature of this method of bootstrapping is that the CBM is implemented as a factored interpreter (or FI, see chapter 3). The concept of an FI has come directly from the thesis research and was initially presented in chapter 3. This means that an IPS built on the CBM as a base has the advantage of interpretive execution, but is able to run at the hardware speed of the host machine.

The procedure for the bootstrapping process is modelled in figure 5C-1, using the notation of Sklansky [Sk 68] outlined earlier in this chapter.

The diagram is rather large and cumbersome, and we will give a simplified model later; but for now, we will describe each step of the bootstrapping process in detail. The steps in this procedure are the following:

creation of correct programs without all the effort normally required to map each problem onto the linear store of such machines;

(2) to raise the level of machines to a point more commensurate and helpful with the common operations done on present day computers using high level languages;

(3) to provide a uniform level of control within which certain classes of operations (such as indiscrimately using arithmetic operations on addresses) are invalid, and in which data structures are recognizable and flexible objects, instead of vectors of binary words with no inherent, consistent structure.

Another consideration voiced by Iliffe is the need for non-linearity and flexibility of program, data and control in the framework of "on-line interaction" — a point which we have used as the key for the development of the CBM, especially with regard to the work preceding this chapter.

One surprising facet of Iliffe's machine is its small number of operators; this can be directly traced to the fact that the BLM memory is structured. It has only one ADD instruction, for instance, instead of the plethora usually found on normal computers. Also, since it can be described in terms of an underlying Von Neumann type machine, implementing the BLM on most computers would be little trouble. This enhances the smallness of Iliffe's machine and increases its use for purposes of bootstrapping and inter-machine system mobility.

There are, however, some respects in which the BLM is either not general enough or is unsuited to interactive programming, and it is these which have been expanded or modified in the CBM design.

FORTRAN processor; the conversion was only made easier if one existed. Even without an available FORTRAN, most of his work has been transferred from machine to machine in terms of only man-days or man-weeks, a notable accomplishment when compared with the problems normally encountered by individuals and companies when converting programs between machines.

The notion of a set of basic primitives in which to build has provided much of the motivation for the CBM. Exportability is inextricably bound up with bootstrapping, because one only wants to export bases on which systems have been constructed rather than transforming entire systems themselves. In our case, this set of primitive data structures and operators has taken the form of a machine design, although a simple language with the same sort of primitives would probably do equally well. The machine framework seems apt because of the present appearance of microprogrammable computers, and because of the level of detail of the CBM's description.

5B1 Iliffe's Basic Language Machine

Another reason for presenting the CBM as a machine is that many of the ideas for it were suggested (and, in some cases were already developed) by Iliffe [I1 68, I1 69] in his description of the Basic Language Machine (BLM). Iliffe's goals, in condensed form, can be stated as the following:

(1) to better define the interfaces between programs and data (as opposed to Von Neumann's concept of instructions and data being indistinguishable [VN 58]) in order to aid debugging and the

for [N]. However, once $\frac{L'}{MM}$ is available, it is presumably not a difficult task to design $\frac{L}{L'M}$ since the code production of these two translators contain much in common $(L' \subset L)$, and this would be reasonable if the language L were to be implemented only on [M].

The important point is that by introducing some extra design work to write $\frac{L}{L'L'}$ for the first implementation, as well as an extra (but automatic) step during bootstrapping, we can more easily move L from machine to machine simply by rewriting L' for the new machine. This is certainly an idealized picture of the issue of bootstrapping and exportability, but it does lend some insight, and we will see the above method II reappear later in the development of a bootstrapping network for an IPS.

5B History of Exportability

Few systems have been developed which even aimed at ease of exportability. The ALTRAN effort [Ge 70], the AED system [Ros 67], and Waite's Base for a Mobile Programming System [Wai 69] are a few which have had this as one of their primary goals. All of them sought to reduce the number of basic primitives needed in order to bootstrap their respective systems. Both the ALTRAN effort and Waite's work described as much of the set of primitives as possible in a standard subset of FORTRAN, the rationale being that most machines have at least FORTRAN supplied with them. Like the AED group they then built a macro system using the set of primitives and proceeded to build the rest of the system on this base.

However, Waite's work was not dependent on the availability of a

by means of assembly language or whatever is not relevant here) so that one had $\frac{L'}{MM}$.

(2) L was written in terms of L' (i.e., in terms of a proper subset of itself) to translate into M, vis. $\frac{L}{L'M}$.

(3) Executing $\frac{L'}{MM}$ on [M] with $\frac{L}{L'M}$ as input yielded $\frac{L}{MM}$, the desired compiler for L on [M].

Diagram 5A-2 below shows this method, which we will call method I:

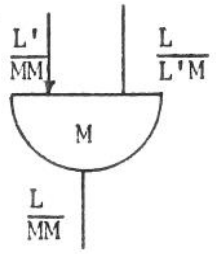


Figure 5A-2: Method I Bootstrapping

Alternatively, in place of step (2) above, one could write $\frac{L}{L'L'}$ and insert it as data in the following way, called Method II:

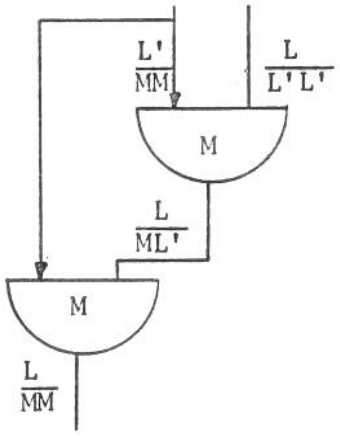


Figure 5A-3: Method II Bootstrapping

Method II has the advantage that only the initial L' has to generate machine code. Thus, when moving such a system to a new machine [N], it is necessary only to write $\frac{L'}{NN}$ since $\frac{L}{L'L'}$ is still valid

translation of the language in which S was originally written into M). The execution of $\frac{S}{MM}$ on machine [M] with $\frac{L}{SB}$, a translation program for L into B, written in S, as its input data is represented in the following diagrammatic way:

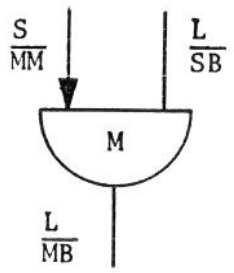


Figure 5A-1: Execution Diagram

where $\frac{S}{MM}$ is the program being run on [M] (the arrow indicates this); $\frac{L}{SB}$ is the input data for the program, and $\frac{L}{MB}$ is the result of running $\frac{S}{MM}$ on [M] with $\frac{L}{SB}$ as data. That is,

$$M\left(\frac{S}{MM}\right)\left(\frac{L}{SB}\right) = \frac{L}{MB} .$$

The exact rules governing such simplifications of "execution expressions" is given in [Sk 68] and will not be further defined here.

Now, the output of such an execution may itself be program or data for another execution, thus forming a network of programs and data. The execution of these programs can be considered to proceed sequentially or in parallel as long as one program has finished before it output is needed by another.

Classically, bootstrapping a language L onto some machine [M] meant that

(1) a subpart of L, denoted by L', was implemented on [M] (whether

5A History of Bootstrapping

Sklansky et al [Sk 68] have suggested a notation for describing translation systems which is also useful for diagramming bootstrapping operations and movement of systems from machine to machine. In this notation, a program is represented by a fractional form such as $\frac{q}{L}$: an algorithm q expressed in language L. A translation algorithm is denoted as $\frac{A}{B}$, standing for an algorithm (not a program) which translates programs written in language A into equivalent programs in language B. And, a program written in L for the algorithm $\frac{A}{B}$ (i.e., a translator program from A to B written in L) is expressed as $\frac{L}{L}\left(\frac{A}{B}\right) = \frac{A}{LB}$. We will also characterize an interpreter for the language A, using the language B for interpretation, and written in L, as $\frac{A}{LB}$.

A program for a machine [M], written in some language S, must have the form $\frac{S}{MM}$ in order to run on the machine: i.e., we must have an algorithm $\frac{S}{M}$ which is itself expressed in M (probably obtained from a previous

5 Bootstrapping an IPS (With an Eye to Exportability)

This chapter deals with the construction details for an IPS. In particular, we give a bootstrapping sequence in terms of what parts must be written and when. Then we will outline a base for bootstrapping an interactive system. It is hoped that keeping this base sufficiently small has also introduced some ease of exporting the resultant system from machine to machine.

This bootstrapping base is presented in the guise of a "machine" called the Conversational Base Machine (CBM); that is, a memory organization and instructions for operating within that framework are developed. Many of the basic constructs which we have used in the algorithms and data structures previously discussed will reappear as primitive (or almost so) in the CBM, and we will attempt to mention those as they are presented.

One thing needs to be stated about the CBM: although it is given here in all its generality, it is not necessarily intended that every IPS will implement the CBM as described. It is a spectrum of machines: at its most flexible extreme, it can handle many of the tasks of an IPS itself, such as type checking, detection of undefined and monitored values, automatic coercion of operands, etc.; at its most rigid extreme, it could be viewed as a set of macros for some machine's assembly language, which macros only really save the size of code written by the implementers. We will describe a "moderate" version of the CBM, taking advantage of the fact that the IPS above it is held responsible for some things, while the CBM checks others.

almost certainly not true of a separate interpreter and compiler for a language (nor indeed for two different compilers for the same language!)

One problem with this is encountered by control statements such as the IF-statement which depend on the execution of some code in order to determine which subnodes to activate. Such routines need to know that they are being called to compile so that they will not attempt to access values which do not exist due to the suppression of execution. Also, any nodes which depend on certain attributes of the values computed by a subnode must be able to expect those attributes to exist despite the fact that no execution occurs; if each such node has this information as part of its description, this requirement is satisfied.

When each node (in the tree to be compiled) has been called, there may still be more than one code string with calls (comp-calls or interp-calls) in other code which uses them. These can then be replaced by the actual code of the referenced node in the same manner as described in section 4C2 (Simple Jumps and GOTO's). Once this is done there will exist only one code string for the program.

One further pass over that code can replace all the indirect relative jumps by direct relative jumps using the [FIRST:LAST] pairs for that code string. At that point, the parse tree can be considered expendable and its storage reclaimed.

Hence, with little extra mechanism a TFI system can also function as a compiler — indeed, an incremental compiler. And since the same routines are used when compiling as when interpreting, there is a good chance that the compiled version of a program will act the same when interpreted, which is

some group of statements or procedures.

4C4A Compile-time Bindings and Assumptions

When the IPS is commanded to compile some program, it could do so without requiring a complete set of declarations for the necessary variables by simply using the default bindings which the TFI would use. It could also require the user to verify each of those bindings by displaying them and requesting approval for each. These are strategic decisions and are best left to individual users to pick their favorites. But what if it is desired that certain variables be allowed to remain flexible with respect to certain of their attributes such as their type? Then it is necessary for the user to tell the system that those variables are to be left unbound — just as it was desirable for the user to give bindings when the system was trying to be flexible and he wanted some fixed bindings.

4C4B The Compiling Mode of a TFI

In one sense we have already shown the TFI behaving as a compiler in the REVERT algorithm. However, simply turning off execution does not make the TFI compile the entire program since not all subtrees will necessarily be executed. In order to make it a compiler we employ the rule that no node is activated to compile until all its subnodes have been compiled. This can be viewed as a depth-first, left-to-right tree walk in the parse tree and will cause each node to generate code. Of course, any node which already has valid code will simply be passed over.

4C4 Compiling with a TFI

Although the TFI mechanisms offer a sizable increase in speed over normal interpretation, it is not as as efficient as a reasonable compiler might be. In the interpretation/compilation spectrum a TFI allows a program to change representation in order to flow between the mentioned extremes. But, like the ancient Greeks who did not understand the regularity (or "bindings") of the stars, the TFI stays within sight of the interpretation coastline in order to be able to respond quickly to semantic weather changes. And, for a TFI to be able to produce good compiled code, it must be able to rely on some set of bindings on the variables and programs involved. The catch, of course, is that there is no law of gravitation of variables, so the TFI must turn to a higher authority for its information. Control over those bindings ultimately lies in the domain of the human user; without some statement of constancy on his part, no bindings can be considered as fixed. Hence, in order to allow the TFI to truly compile a program, the user must make a contract with the IPS not to change that program or the variables on which it depends, while the program is active.

Given this contract the IPS can feel free to discard almost all the hooks necessary to allow re-interpretation, including the parse tree, almost all dependency chain elements, multiple code buffers, and indirect relative jumps. Other information such as the symbol table and the program text might be relegated to secondary storage. For simplicity, this contract could be made via a simple statement such as COMPILE β, where β is the description of