INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again - beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106
RICE UNIVERSITY

Language Directed Computer Architecture

by

Larry N. McMahan

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

Thesis Director's Signature

Edward Alvin Feustel

Houston, Texas

May, 1975
ACKNOWLEDGEMENTS

The author would like to gratefully acknowledge Dr. Edward Feustel for the vast store of knowledge which he has imparted, and for serving as Thesis Director. I would also like to thank Dr. Sigsby Rusk and Dr. Ken Kennedy for serving on my Thesis Committee.

This work would not have been possible had it not been for Scott Warren and his many enlightening remarks on high level language design.

Finally I would like to thank Breck Stewart for her typing and Cindy for her patience and encouragement.
CONTENTS

Chapter I: Introduction to Language Directed Machine Architecture 1
Chapter II: Development of Linguistic Primitives 16
Chapter III: A Dynamic Process Model 49
Chapter IV: A Computer Architecture 73
Chapter V: An Evaluation and Summary 105
Chapter I

Introduction to Language Directed Machine Architecture

This thesis is concerned with the use of language directed computer design. If we view the basis of computer activity as the statement and solution of problems for human needs, the design of systems to effectively accomplish this forms an active research area of computer science. If we take the philosophy that the complexity of a task for which we can propose a solution is limited by our intellectual capacity to grasp the problem and its subsequent answer, then machines should be designed and built with the goal of minimizing the repetitious and mechanical steps involved, allowing the individual to concentrate on conceptualizing the task. This is evident today in a wide selection ranging from hand calculators for household use to giant systems for real-time weather modeling and other complex applications. Thus, if we are to submit larger, more complex tasks to the computer, we must have a clear ideal of how a task appears.

If we consider a task of the kind submitted for machine computations as an abstract process, we see that
it has several components: 1. Formulation of the problem in a medium which models the individual's thought process; 2. Translation of the problem, usually by both man and machine into a formal logic (machine code) for computation; 3. Interaction of the man and machine to produce a workable solution; 4. Solution of the problem by the machine system; 5. Translation of the results into a form understood by the user; and 6. Evaluation of the results by the user. In viewing a task from posing to solution, we note three areas in which work must be done. First, there is the conceptualization medium in which the man poses his task, evaluates the results, and from which viewpoint he interacts with the machine. Second, there is the machine code to which computation is applied. Finally, between the two there is a two way translation process, the burden of which is shared by the man and machine. Thus, if we are to admit the solution of larger more complex problems, we must minimize the work required in each of these areas. We wish to:

1. Recognize a programming medium which provides an accurate model of task conceptualization.

2. Provide a machine which can solve a complex task at a minimal cost of space and time.
3. Devise a translation process which is like a mathematical identity to the machine and a conceptual identity to the user.

The question of task formulation is concerned with providing an algorithmic model which is at the same time a well defined formal (logic) structure and an accurate description of the thought process involved in posing and solving a problem. In current programming practice it has lead to such activities as the study of grammars in formal automata, and to the formal description of programming languages. In this vein, the need to represent mathematical formulas was recognized early and led to the development of FORTRAN [1]. Since, ALGOL60 [2] has provided a linguistically concise language and dealt with such concepts as program structure, name context, and recursive evaluation of functions. Other work in programming languages, eg: COBOL, LISP, ALGOL68, or APL [3,4,5,6] has dealt with such a variety of topics as structured data, binding, modes and complex operators. In each case the thrust has been to develop linguistic structures for task formulation, thus relieving the use of this burden.

Being the physical basis for computing activity, the question of machine design was, historically speaking, the
earliest problem to be recognized. Early computing machinery was designed to relieve the user of large and repetitious numerical computations [7]. For this reason machine design originally proceeded along the line of large calculators. As linguistic knowledge became more sophisticated machine design proceeded in parallel. Thus, from the early von-Neumann machines we have seen the development of pure procedure code, hardware stacks, segmentation, multilevel addressing, and mode tags [8,9,10] as the languages using the machines grew more complex.

The translation process must bridge the gap between the programming medium of task formulation and the computational machine language. Whereas early programming media consisted of little more than a symbolic representation of the actual machine code, the question of translation is currently being attacked both in the area of compiler design and the area of process modeling. Strictly speaking, process modeling can either be considered an aspect of formulation if a formal language model is represented, or machine design if the model is a logic model. In any case, the objective is to provide a clear and simple translation between the linguistic constructs and their machine language counterparts.
The techniques elucidated in the proceeding paragraphs have sufficed to bring us from the machines designed to aid simple repetative numerical calculations to the large complex multitask systems presently available, yet they suffer several drawbacks. Consider first the initial translation (compilation) process. In conventional machine architectures, in particular those with such features as linear instruction sequencing, undifferentiated memory, absolute addressing, and the branch instruction, the compilation process involves the recognition of the treelike block structure in the linear input text of a high level language, then its transformation into an array of instructions for execution by a sequential processor. This process is disadvantageous because the conventional architecture does not adequately represent the structure of the program and the operations do not correspond in a uniform functional manner. That is, a particular language construct, may have several alternatives for code expansion; similarly a particular operation code may be used in the expression of several language constructs which are unrelated, syntactically or semantically.

An example is the occurrence of the operator "+" in a programming language. In this instance there is a physical
(hardware) ADD instruction which is the machine representation of that operator, but there is usually no direct equivalent to many structural concepts such as iteration, conditional evaluation, context (block) entry and exit, and function invocation, and these must be transformed to (possibly varying) instruction streams, or in some cases to calls on operating system routines. But even in the case of our simple ADD instruction it may not be that easy. The IBM System/370 has fourteen ADD instructions [11], all of which may not be used to implement a particular language. Further, the instruction used will depend on the particular usage of the +. Hence, it is necessary to design a great deal of redundancy into both the machine and the language to achieve a desired level of generality. As a result, the compiler for a given language becomes correspondingly larger and more complex.

Consider next the interaction of the man with the machine to produce a solution. On a conventional machine this requires the user to recognize the semantics of output and runtime diagnostics and appropriately modifying his program to obtain the desired results. The disadvantage of this is that often the semantics of the diagnostic does not correspond directly with the programming error that
elicited it. In fact, a particular error might produce any one of several runtime responses, and one diagnostic might not be able to distinguish among several possible errors. Also, the lack of direct correspondence requires that the user, beyond understanding the language, must also understand the implementation and be familiar with the workings of the machine itself. Several concrete examples again illuminate the question.

A relevant example is concerned with the occurrence of error U4000 in the use of PL/1 on the 370 [12]. This diagnostic indicates that a program interrupt has occurred during PL/1 error handling routines. This is enough to indicate that a disastrous error has been made, but not enough to give any clues to the semantic interpretation of the offending statement. Hence it would seem that the answer is to improve the linguistic relevance of the diagnostics. But even a "linguistically relevant" diagnostic such as the PL/1 subscript range message displays certain difficulties. In the case of the multiple occurrence of the array in the statement indicated (ie.: A(I,J)* A(J,K)), an assembly listing of the load module and a dump is required to determine the name and value of the offending index variable without an additional machine run. This is
because the actual occurrence of the error is given as an offset in the load module. Hence, beyond an understanding of the programming language itself, the user is required to be familiar with the machine design and the implementation constraints it requires.

Recapitulating, we note that we began with the abstract but almost simplistic notion of reducing the mechanical and repetitious steps involved in a task, allowing a machine user to concentrate his efforts on conceptualization of the problem. Taken directly this is a difficult goal to define, much less accomplish. Nevertheless, by examining the nature of computational tasks and the structure of the machines designed to solve them, we have been able to transform our simple abstract notion into several practical goals for improving computational systems. They are:

1. Provide more powerful linguistic constructs in hardware.
2. Reduce compiler complexity.
3. Provide a machine favorably comparable in size and speed to present design.
4. Reduce interactive complexity.

Thus armed with these practical translations of our abstract
notion we consider the question of how should a machine be designed to best achieve these goals.

The design problem of how to build language directed machines can be approached by considering the analogous usage question of how to effectively program tasks for machine computation. This has been the question recently probed in the area of structured programming and has resulted in such common sense dictums as task modularization and top down programming. But these provide only the methodology of efficient programming practice. If we are to understand the programming process, we must grasp the underlying concepts. As Dijkstra [13] points out, programming is the art of stepwise refinement of intellectual problems (in their initial refinement) to versions tailored to be run on the available machine (in their final refinement). We see that this is in fact the case and that the number of refinements required is directly related to the programming media available. That is, the process is one from thought to algorithmic specification (possibly in the programming language provided) then through any recognizable intermediate forms (trees, assembly code, etc.) to the running machine implementation. It follows then that the architectural design for such a machine proceeds effectively along this
course, providing the definition (metalanguage, if you will) for the program writing process. Thus we are concerned in turn with the specification of the linguistic constructs employed, the exposition of a computational model for their behavior, and a machine implementation of the model. McKeeman [14] supports this view when in arguing for machines which can more effectively do our bidding he points out that "We are our own best model for computer organization."

Chapter II is devoted to the specification of the programming constructs to be employed. To do so, the psychological and linguistic aspects of programming are considered. Psychologically, programming is concerned with the transformation of a problem as a thought process to a programming language specification. That is, the structure of the algorithm, the effect of the transformations, and the meaning (interpretation) of the linguistic primitives. Linguistically, these areas are in the order of increasing semantic and decreasing syntactic complexity. At the program level they correspond in turn to statement (clause) structure, expression evaluation, and variable allocation and reference, respectively. Each of these areas is considered and a description is made of the linguistic constructs employed. A control structure for the modularization of tasks
is presented based on the results of structured programming, and specifically on the work of Dijkstra [13,15] and Mills [16]. A functional unit using modes and a two level syntax is developed similar to that of ALGOL68 [5] or other "typed" languages. Finally an allocation binding and environment management mechanism based on block structured languages and specifically an extension of previous work with MADCAP VI and tagged architecture [17,21] is presented.

In Chapter III a computational model is presented based on the linguistic constructs developed in Chapter II. Like the contour tree model of Johnson and Berry [18,19,20], it consists of an algorithm and execution record. The algorithm is a static region which models the syntactic structure of the program. Unlike the contour tree model and most other models which deal primarily with the semantics of data flow and present the algorithm as simply a copy of the program text, the present model gives a more detailed description of the algorithm, presenting it as a read-only memory region structured according to the linguistic constructs developed in Chapter II. The execution record consists of a set of access environments a la Johnson and in addition a locus of control. A linked stack
implementation is given for the access environments, with environment retention providable through heap storage allocation on block exit. Usually control locus is modeled as a pointer into the program text (algorithm) while "remembering" the call sequence. In the present model however, the control locus is modeled as a stack of statements within whose scope the particular machine state exists. Thus block exit and return from called functions is made explicit in the model as is the semantics of statement flow. Finally, a process indicator is defined, which is the current access environment, control locus, and temporary values available at some specific time during partial evaluation of a program.

The model presented in Chapter III is next used to serve as the blueprint for the register transfer machine design developed in Chapter IV. The algorithm region is implemented as a read only descriptor to a memory segment containing the program algorithm. The execution record is implemented as a descriptor to a workspace memory segment containing the access environments, control locus, and expression values of the dynamic process. The set of access environments is implemented as a linked Name Stack with the current environment addressed by the Environment Base
Register. The control locus is implemented as a Control Stack with the Instruction Register as the top element. The value of an expression is represented by the Temporary Register as the top element of the Operand Stack. In addition to the register transfer description a simulation is also presented at that level via a BCPL program. This program serves a twofold purpose in completing the description of the machine. First, it provides a resolution to any questions regarding semantic interpretation of register usage. Finally, it yields a dynamic representation of what has heretofore been a static description. That is, it considers the question of how does the machine run.

This question is dealt with in detail in Chapter V. We wish to show that the machine design presented in Chapter IV as a stepwise refinement of programming linguistics satisfies the objectives of reducing task complexity. From a programming viewpoint we wish to facilitate the implementation of linguistic constructs as machine instructions and provide a diagnostic semantics which preserves the classes of meaning established by the programming language. From a systems operation viewpoint we wish to simplify both the compiler writing and the program
compilation process. At this point a slight digression will be made to indicate how such a high level machine would be better suited to this particular objective than conventional machines. Recalling our discussion of the compilation process, we note that on conventional machines this involves two major passes: the recognition of the program structure in a linear input stream (parsing), and the production of a sequential array of instructions (code generation). Since the register transfer machine presented operates on a program structure resembling the intermediate tree of a conventional compiler, the computational overhead associated with code generation is effectively removed, both in terms of a program that must be written by the compiler designer, and the cost in time and memory space to the programmer. Finally, if our implementation is to prove of positive value as a computational tool, the foregoing advantages must be realizable without degradation in either speed of performance or memory utilization when compared to conventional machines. We would in fact expect to do much better than to merely break even, since reducing mismatch and redundancy in the machine implementation of language primitives allows fewer instructions of a comparable complexity to those on conventional
machines. To ascertain data in this area example program simulations are compared with programs to perform the same task written in PL/1 or BCPL for the 370/155. Finally, the results of the simulation are presented and suggestions for improved machine design are considered.
Chapter II

Development of Linguistic Primitives

This chapter is devoted to the first step in the refinement of a high level language directed machine, the development of the linguistic primitives with which a problem solution is expressed. This step parallels the programming question of representing the conceptualized solution to a task as a formal specification (initial program refinement). Thus we are concerned with examining the psychology of task formulation and recognizing the linguistic constructs which most effectively represent the proposed solution as it is conceptualized. If we consider the psychology of task formulation we are concerned with the following: a) the meaning of elements within the universe of discourse, b) action which causes change within the universe of discourse, and c) the form of a solution to a task as an aggregation of solutions to subtasks or items from the universe of discourse.

Each conceptual category has a corresponding area of linguistic significance. At the level of meaning we
consider the semantic interpretation of the linguistic primitives with which we deal. Thus we are concerned with the representation and interpretation of values and the structure of data and the result (value) of operators and statements (clauses). Action corresponds to the linguistic notion of a change within the universe of discourse. Hence in this area we are concerned with the creation and deletion of names, the acquisition, referencing, and disposition of values, the selection of subsets of multiple values, the effects of transformations and functions, and finally the choice of statement sequencing. Form is linguistically denoted by the syntactic structure of the program algorithm. The corresponding topics of interest are then the representation of basic tokens, and the expression structure available within the linguistic framework. Thus it is possible to define an effective linguistic representation of the conceptual categories noted. Such a representation would be composed then of a) a family of allowable values from which a given program would construct a particular universe of discourse, b) a set of formulae applicable to all meaningful universes of discourse which yield a new (possibly the same) universe of discourse, and c) a set of clauses (statements and formulae) from which
a given program algorithm is constructed. Each of the above linguistic areas is examined and the corresponding primitive constructs delineated.

Consider first the representation of elements within the universe of discourse as linguistic values. Each element may denote a primitive value (a particular number, truthvalue, function denotation, etc.), an aggregation of values (rows and structures), or a named entity which may be attached to a particular value by binding. Thus each element can be viewed as belonging to a particular class of values and having a particular denotation within that class. Thus a value is composed of two fields, a mode and a denotation. The mode of a value identifies the contextual class to which it belongs and the denotation yields a particular item within that class. It remains then to delineate the meaningful modes along with the semantic interpretation of their representations.

A primitive value is either an element of a data mode or an element of mode proc. The data modes and their respective representation sets are given in Table II-1. The modes int, real, and cmpx represent the common notion of arithmetic numbers. Bool denotes formal logic. Char allows the use of text in a particular alphabet. The mode
**user** is a representation of Hoare's notion of enumerated
data types defined by the user [22]. The mode **mode** allows the use of a context class (modename) as a value. Finally, in a categoric definition system it is useful to denote the "non-entity" which is a member of none of the classes, the mode **null** serves this purpose. Note that the denotation of an element of a data mode has semantic interpretation which depends only on its representation. The final primitive value is an element of mode **proc**. An element of mode **proc** is a procedure closure. Its denotation is an ordered pair of elements, a function denotation and an evaluation environment. In all cases a function denotation is the location of the defining algorithm. Its semantic interpretation is determined by the segment of program code at the given location, and is constant for the duration of the process. As a constant the evaluation environment value is a literal denoting the number of activation levels it is above the active environment. As a variable it is a descriptor to the proper environment. Its semantic interpretation is the particular universe of discourse (subset of possible values) available in the evaluation environment. Primitive values provide the terminal nodes of the linguistic tree.
Aggregations of values are linguistically represented as rows of values or structures of values. A row of values is a homogenous sequence of values (possibly varying) with variable subscripts. A structure is a (possibly nonhomogenous) constant natural \((1,2,\ldots)\) sequence of values. Both are represented by elements of mode \textit{row}. The representation of an item of mode \textit{row} is given in Figure II-1. A structure is denoted as a row of mixed values with an initial subscript of one. The semantic interpretation of a row is the value of its elements if there are any, otherwise it is null.

Thus we have developed the sets of values available in a universe of discourse. In order to allow for the referencing of these values within a particular universe of discourse the linguistic notion of the \textit{variable} is used. A variable has three attributes, a \textit{name} which is the program appellation (identifier) to which the value is attached, the \textit{location} at which the value resides, and the \textit{value} itself which provides the semantic interpretation of the variable. Names and locations are represented by elements of mode \textit{name} and \textit{loc} respectively. Thus not only may we name values, but the names themselves may in turn be values.

The preceding paragraphs have presented a linguistic
representation of values, names of values, and locations of values (which may in turn be values) as elements of a universe of discourse. At this point we wish to consider a set of actions which perform computation by acting as a transformation on the set of universes of discourse. Such an action is linguistically represented by a formula. A formula returns a value as its result. In addition it may do any of the following: a) add names to or remove names from the universe of discourse, b) obtain or alter the value associated with a name, c) input or output a value, d) perform basic data transformation (arithmetic, etc.). Thus we are concerned with the expression of a formula and an evaluation mechanism which produces the result. In its most general form a well formed formula is a tree of operators with operands at each of the leaf nodes. For implementation purposes postfix polish is taken as a canonical form and the formula is represented as a descriptor to a sequence of operands and operators in that form. An operand is either a primitive constant (data mode or mode proc) or a variable name. The operand is stacked in temporary working storage (and in the case of mode proc the environment is bound to the function designator) and the next item in the sequence is examined. When an operator of
n arguments occurs the top n elements of the stack are
taken, the operation is performed, and the result is placed
on the stack. If at any time an operator occurs and there
are not a sufficient number of values on the stack, or if
the formula terminates with other than one value on the
stack, then the formula is not well formed and a semantic
error is detected. When the formula terminates the single
element left on the working stack is taken as its value.

Operators may be linguistically divided into six
classes: 1) referencing*, 2) transput*, 3) rowing*,
4) calculation*, 5) assignment, and 6) function invoca-
tion (call). Those classes marked with an asterisk consist
of a number of operators and may be further subdivided.
Classification of operators is shown in Table II-2. Each
operator provides one of the basic transformations from
which a formula may be built. The question of the opera-
tors' linguistic relationship (precedence, etc.) is one
that is left up to the discretion of the compiler writer
by the use of the postfix notation. A detailed discussion
of the various classes of operators will be presented in
the following paragraphs, giving the value and effect of
each operator.

To understand the individual operators the structure
of the universe of discourse must be examined so that their effect on it will be clear. Since the semantic units of a formula are its operands, a specific universe of discourse is the set of values known within a particular language and the set of names known in the environment in which that formula is evaluated. Two methods of associating names and environments is considered. In the first case names that appear in the evaluation of particular formula appear as variables in a context of that formula. A context is the nested set of environments surrounding the formula. Thus each environment consists of the set of names defined at that level and a link to its surrounding environment. The outermost environment contains the global variables of a process and links to the system library. In addition to contextual binding a dynamic binding method is considered in which variables are created and destroyed by the programmer. A reference to a dynamic variable is semantically interpreted as the most recent creation of that name which has not been freed. If a reference mode operand (i.e. `name` or `loc`) occurs as the argument of a primitive operator requiring a data mode element then the valuate operator (see description below) is applied to the argument of the operator until a non-reference mode value is obtained. This
value is then taken as the argument of the operator and must fall into the proper semantic class (mode). Further, the actual modes `name` and `loc` are considered representations of the semantic class of references. Thus the modename `REF` denotes a variable whose actual value during evaluation is an item of either mode `name` or `loc`. Thus we have a description of the universe of discourse and the accessing of its elements. This in turn gives us a concise description of the operands used in formula evaluation. In the following paragraphs the operators will be examined according to the classification given in Table II-2.

The first class of operators that we consider is referencing operators. These are concerned with the creation and deletion of variables in the universe of discourse, and with accessing their attributes. The operators are `take, heap, free, locate, and valuate`. The `take` operator creates a new unique name at the first available location of the current environment. There are a variable number of arguments, according to the mode of the variable being created. If the first argument is an item of mode `row, loc, or name` an element of that type will be allocated and it will be initialized to the value of the first argument. Otherwise the mode of the first element must be `mode`. That
is: the first element must be a modename, see Table II-1.b. If the value of the first argument is the name of a primitive mode the an element of that mode is allocated and initialized to the default value given in Table II-1.

If the value of the first argument is the modename REF then a variable is created with mode loc and is initialized to null. In the above cases the operator is unary and the value of the operation is the location of the variable. If the first argument is the modename ROW then there are either three or four arguments. The second argument must be an item of mode int. It denotes the upper subscript of the sequence of elements. If the third argument is of mode int then it denotes the lower subscript of the sequence of elements. Otherwise the lower bound is taken as 1 and the third is taken as final argument. The last argument must be either a mode name or an item of mode row. If it is a row item upper - lower additional copies of the structure descriptored by the row element is made and a row of elements is allocated which are the descriptors of each of the copies. If the last element is a mode name upper - lower + 1 locations are taken and the elements are initialized to the values in Table II-1. In all cases when the first argument of the take operator is the mode name ROW the value of the operation is a descriptor to the row of
elements allocated. Finally, the first argument may be the modename STRUCT. In this case the second argument must be a positive \texttt{int} denoting the number of elements in the structure. If the number is \texttt{N} then there are \texttt{N+2} arguments, the last \texttt{N} being the modename or row descriptor of each of the elements. The value of the operation is then a row descriptor of mixed elements indexed from 1 to \texttt{N}.

By nesting take operators multi-row structures may be created. Any variable existing within a particular environment is accessible for the lifetime of that environment. The take operator provides for the allocation of variables with contextually defined names.

The second referencing operator is heap, which provides for the allocation of dynamic names. Like take, there are a variable number of arguments. If the first argument is a primitive mode name, a \texttt{row}, \texttt{name}, or \texttt{loc} there must be two arguments of which the second is a heap name. There is an associative name table associated with the process. The variable is allocated at the first free location beyond the last occurrence of the same heap name and initialized like take. Otherwise an aggregation of elements is taken in the heap and the value of the operation is a descriptor to the elements. A variable allocated
by a heap operator is dynamically accessible until deleted by a subsequent free operator. A free operator has one argument. If it is a name the locate operator is applied. If the resulting argument is a heap loc then the variable in the namelist occupying that location is deallocated. If the argument is a heap row then the corresponding sequence of elements is deallocated. Otherwise the argument must be a loc or row of the most recently allocated item in the current environment. The operation returns a null value.

The above operators implement the creation and deletion of variables within the universe of discourse.

Of paramount importance in the evaluation of formulas is transformation of the names and locations of these variables into the subsequent locations and values. The last two referencing operators locate and valuate serve this purpose. Both are unary operators. The argument of locate must be a name or a loc. In either case the value of the operation is the location of the variable; ie. in the case of a loc the result is merely the argument. The argument of valuate may be any mode. If it is a name or a loc the corresponding variable value is obtained; otherwise the result is the argument. Thus locate and valuate complete the description of the referencing operators.
The second class of operators is the two transput operators, read and write. Read takes a single argument which must be a modename. The current item at the front of the input queue must be an element of that mode. (In the case of mode name REF the item must be a name.) The element read is the value of the operation. Write requires a single argument which may be an element of any mode. The argument is placed on the output queue and also becomes the value of the operation. The two above operators provide for the acquisition and disposition of data by a task.

The third class of operators that we consider is rowing operators. These are concerned with the manipulation and accessing of rows of values. There are six operators: slice, index, list, length, upper, and lower. The slice operator requires two arguments. The first argument must be some row of mode (for example row of int). The second argument must be either int or row of int. If it is int then the operation is subscription along the major axis. The result is the element of the row in the first argument indexed by the second argument. If it is row of int then the operation is compression along the major axis. The result consists of a row of those elements whose indices appear in the second argument in the order in
which they appear. The second rowing operator is index. It requires two arguments. The first argument must be row of any mode. The second must be int or row of int. The length of the second argument, N, must be less than the number of dimensions of the first (ie. the first must be at least row of ... row of mode, N times). Subscription is performed along the first N dimensions (major axes). The result is a loc referencing the chosen element. The third rowing operator is list, which requires N+1 arguments. The second argument must be a positive int N. N more arguments are taken as elements of a row. If the N arguments are of the same mode then the result is row of mode, otherwise the result is row of mixed. Thus list provides for the transformation of a set of values into a single structured value. The last three rowing operators are all unary and all provide subscript range information. All require an item of mode row. Upper returns the maximum possible subscript. Lower returns the minimum possible subscript. Length returns the number of elements in the row; that is, upper - lower + 1. The mode of the result is int. All rowing operations produce no side effects, but return a value only.

The fourth class of operators is calculators. This
class is further subdivided into logic*, precedence*, conformity, arithmetic*, translation and identity; where * indicates multiple subclasses. Before the individual operators in each subclass are discussed the general semantics of calculator evaluation is examined. Calculators, like rowing operators, have only a value and produce no side effects. Calculators are either monadic or dyadic. In either case the argument evaluation is the same: If any argument is either a name or a loc then evaluate is repeatedly applied until a nonreference mode is obtained. If all the arguments of a calculator are null the result is null. If the dereferenced argument of a monadic calculator is a primitive element whose mode agrees with the operator, then an element of the result mode is produced. Otherwise the argument must be a row of the proper argument mode. The operator is then applied item by item and a row of result mode is produced. In the case of dyadic operators again scalars return scalars. A pair of row elements must match subscript bounds and the operator is applied item by item. If either argument is a scalar and the other a row the scalar is expanded and applied item by item to form a row. If either argument is a row and the other is null then the operator is applied as a reduction
and the result is a scalar (eg.: addition becomes summation). The above discussion encompasses the limits of allowable mode usage with operators. In particular the use of a scalar and null with a dyadic operator is not semantically meaningful. The proper modes of domain and range for the subclasses of calculator operators is given in Table II-3.

The first subclass of calculators is the logic operators. There is one monadic operator which is logical negation (not). There are three dyadic operators, and, or, an exclusive or (xor). In reduction they become all, any, and odd respectively. There are five precedence operators, all dyadic. They are less, notgreater, equal, notless, and greater. In reduction they become increasing, nondecreasing, same, nonincreasing, and decreasing respectively. The third subclass is a single conformity operation. It is dyadic and the result depends on the mode of the arguments, but in all cases the result is a bool. If both arguments are of mode mode (ie. modenames) then the operation is test for equality and yields true or false. If only one argument is a modename and the mode of the other argument is a representation of that mode the value is true, otherwise false. For all other modes the value is true only when both
arguments are the same mode. The fourth subclass is the
arithmetic operators. There are five monadic exponential,
arithmetic operators, exponential, logarithm, norm, nega-
tion and conjugate. If the argument of conjugate is either
real or int the result is negation. In all cases the mode
of the result is the mode of the argument. The remainder
of the arithmetic operators are dyadic. It is recognized
that arithmetically the integers are contained in the reals
are contained in the complexes. Thus real + int becomes
real. In reduction subtract and divide become alternating
sum and product respectively. The next subclass is the
translate operator. It has two arguments. The first is
any value of mode row of bool, char, user or int. The
second must be one of the modenames BOOL, CHAR, USER or
INT. The result is the equivalent representation of argu-
ment one in the mode of argument two.

The fourth operator class is the assign operator.
It has two arguments. The first argument is the destination,
it must be an item of mode name, loc, or row. If it is an
item of mode name then locate is applied. The second argu-
ment is the source. If the source is a name or a loc and
the destination is loc name, loc loc, or row of ref then the
assignment is made either as a scalar or by expansion into
a row of elements. Otherwise the source is dereferenced. At this point if the source is a proc or data mode then the destination must be loc (source mode) or row of (source mode) and the assignment proceeds as above. If the source is a row of mode and the destination is a loc row the row descriptor will be assigned. Else the destination must be a row descriptor which agrees in mode and subscript range with the source. Assignment is then performed item by item. The result of an assignment is the value of the source.

At this point the structure of the universe of discourse becomes active in the description of the assign operator. The existence of name, loc, and row values with their inherent location references require environmental control over their assignment. There are several strategies available. The most restrictive is to prohibit assignment of such variables outside of the current environment. The second is to mark the variable or sequence of elements referenced for retention at block exit. Third, maintaining a reference count for all variables would obviate the need for checking at assignment, but would require reference management techniques beyond the scope of this work. The alternatives will be examined when defining the process model of the following chapter.
The final class of operations is again a single operator call. Call has a variable number of arguments. The first argument is dereferenced and must yield a proc. A proc variable consists of a function designator to a routine of N arguments \( (N \geq 0) \) and an environment link to the next most global environment of the routine. A new environment is created and linked to the proper contextually containing environment. The next N arguments (2 through \( N+1 \)) are taken as the values of the first N variables of the new environment and N new names are created. The partially evaluated formula using the function value is saved along with return information. The routine is then evaluated as a subtask of the calling formula. Subtask evaluation is the subject of clause description given in the paragraphs below. This operation creates whatever side effects are in the function routine and returns its value on top of the working stack of the calling environment.

The preceding paragraphs have presented a linguistic description of operators and operands in the evaluation of formulae. Thus we may now view a formula as a state transformation on the universe of discourse. Looking back to our first refinement of a task we recognize formulae as the transformational entities from which aggregate solutions to
complex problems are produced. Thus we are now concerned with clause structure and the manner of its evaluation. A task solution is a clause to be evaluated in a new environment contained directly within the system environment. A clause is either a statement or a formula. Statements provide for the modular decomposition of clauses into simpler subclauses, and finally at the most basic level into formulae. A statement consists of a statement kind identifier and a reference to a list of arguments. The various kinds of statements and their proper arguments are given in Figure II-2. Thus the program algorithm which is the solution to a task is a tree of statements with formulae at the leaf nodes. There are six kinds of statements. They are while, repeat, if, case, sequence, and exit. Note that the exit statement is also a leaf node. Just as formulae, each statement returns a value. Thus statements may be used as operands. The evaluation of each kind of statement is presented in the following paragraphs.

Looking at results in the area of algorithmic construction we see that the basic building blocks of an algorithm are iteration, selection, and sequential action. To this we add termination as a model of real time control. While it is true that they are semantically equivalent in
the context of the other kinds, one or the other may be
more concise for a particular construction, hence both are
provided. Selection is provided by if and case. Again,
while if is semantically sufficient to represent selection,
an n way choice is often the most concise representation.
This is provided by case. Sequential action is represented
by the sequence statement. It causes the ordered evalua-
tion of n clauses. Termination is provided by the exit
statement. N levels of nested statements are terminated.
Hence we have developed a linguistic classification of
each statement kind.

The first is the while statement. It has two argu-
ments, both of which are clauses. The first is evaluated;
it must return a bool value (or a semantic error is sig-
naled). If the value is true, the second argument is
evaluated, the first argument is evaluated, and the test
is reapplied. If the value is false the statement is ter-
minated. The value of the while is the last value of the
second argument if it has been evaluated. Otherwise it is
null.

The second statement kind is repeat. It also con-
sists of two arguments, both of which are clauses. Both
arguments are evaluated. The second must return a bool.
If it is false then both arguments are again evaluated and the test is reapplied. If it is true the statement terminates. The value of the repeat statement is the last value of the first argument.

The third statement kind is the if. It requires three arguments, all of which are clauses. The first is evaluated and must return a bool. If it is true then the second argument is evaluated, otherwise the third is evaluated. The statement then terminates. The result of the if statement is the value of the last clause evaluated. That is an if statement with a null else clause whose predicate evaluates to false will return that as a value. The case statement is the generalization of the if to a N way selection. There are N+2 arguments. The first argument is a literal positive integer denoting the number of branches in the statement. The remaining N+1 arguments are all clauses. The second is the predicate. It is evaluated and must return an int value. If it is outside the range 0:N-1 it is replaced by 0. Based on this value one of the N remaining arguments is evaluated and the statement terminates. The value of the case statement is the value of the evaluated clause.

Sequence is the fifth statement kind and provides
ordering of clause evaluation. There are $N+1$ arguments. The first is a literal positive integer denoting the number of clauses in the sequence. The remaining $N$ arguments are the clauses, which are evaluated in order and the statement is terminated. The value of the sequence is the value of the last clause executed.

The last statement kind is exit, which provides for abnormal (early) statement termination. It has a single argument which must be an operand dereferenceable to a positive element of mode int. During the evaluation of the arguments of a statement before its termination the processor is said to be within the range of that statement. Thus the position of the processor at a given formula is represented as a nested list of statements within whose range the processor currently is. The purpose of the exit statement is to terminate exactly $n$ levels of statement nesting and resume evaluation at that point. The value of the exit operation is the value of the clause immediately prior to the exit.

The preceding paragraphs have presented a linguistic description of statements in the construction of algorithms as the solution to tasks. Thus we may view a program algorithm as a modular structure of statements and formulae.
With this statement the description of the third area, the form of the program algorithm is completed. Thus we have completed also the original goal which we set out to accomplish at the beginning of the chapter. That is, we have a linguistic description of the psychology of task formulation. In the following paragraphs we wish to examine this linguistic description and see what special properties it may possess. Several areas are examined in detail.

The first area we wish to consider is proof of correctness. This area is concerned with answering the question: how do you know it will work? The object is to provide a convincing proof. Effectively any program may be proved correct for a given set of inputs, but the complexity of the proof varies as the complexity of the program. This is because proof proceeds by induction on the structure of the program. Thus the linguistic description should provide the simplest inductive program structure which spans the algorithm space. Results in structured programming [13, 16 etc.] indicate that the three basic structural units are sequencing, selection, or iteration. Each has a single entry, a single exit, and performs a basic algorithmic command. The single entry is the trick to maintaining inductive simplicity of the algorithm, however by
allowing multiple exits in a uniform manner and more complex algorithmic commands the linguistic power of expression may be increased without increasing inductive complexity. In particular, if a structural unit has multiple exits, each path should either terminate or go to a collecting node from which all exits from that unit proceed to the next unit. Armed with these specifications we develop the statement structure indicated.

Let us consider first the iterator. The test may be performed either before or after the body. Either is semantically sufficient. In practice both are used and in fact a particular construct might be more concisely represented with one order or the other. Hence both are required to avoid node splitting. Thus we have our first two statements, the while-do construct and the repeat-until construct. Consider next the concept of sequencing. Strictly a sequence is a way of ordering two elements, this is extended to the practical usage of a sequence of $n$ elements. This gives us a third statement. The last structural class is selection. Classical two way selection gives us the if then else construct. Again we wish to generalize a selection on the first $n$ integers, or zero for out of range. This gives us the case-in-out construct.
Thus we have two additional statements. The set thus far elucidated is sufficient for algorithmic expression, however the full expressive power allowed by multiple exits has not been fully utilized. We wish to admit exit from within a nested range of statements to an arbitrary level n. Thus we have the exit statement. Thus we have developed a set of algorithmic primitives which maintains the simplest inductive structure for proof of correctness and which has a maximal linguistic power.

The second area involves the question linguistically speaking of what is a variable. Since task solution involves the computation of values in a predetermined fashion, at some point the concept of remembering a value becomes a part of the abstraction process. A variable is the formalized notion of a remembered value. This is accomplished by attaching to the value a name and providing a memory which allows subsequent appearances of the name to be transformed into the corresponding value. At this point note that we have not taken the programming language notion of a variable as a named entity whose value can change, but merely as a value to which a name is attached. If we view the assignment operator as replacing a variable with a new variable with the same name, then the two views are unified. To
clarify this notion we now wish to consider the memory process by which variables are retained.

If we take the representation of a remembered value as an entity that in some sense takes up space, then the problem becomes one of finding a location where this representation may be placed and providing an effective map between the set of names and the set of locations. If we allow names and locations to be values we have created a new semantic class whose value is a reference to some element of the universe of discourse. Thus corresponding to the actual (representation) modes `name` and `loc` we have the logical mode `ref`. Thus a variable whose mode is `ref` will have a value which is either a `name` or `loc`. Further if we allow a semantic class (mode) whose values are names of the semantic classes allowable (modenames) then it is possible to provide allocation as a basic operation within the domain of a formula. The variable attributes name, location, and value may be transformed in that order in a process which programming semantics recognizes as binding. Thus locate and valuate provide explicit binding operators. Thus again by considering the linguistic primitives available we have obtained a coherent semantic description of variables.
This chapter has presented a linguistic description of task formulation. Detailed description of the implementation model has been avoided as this is the subject of the following chapter. It is impossible to identify where each linguistic construct came from, however, readings in the following areas influenced the final description: linguistics and linguistic psychology [23,24]; formal language specification [5,25]; programming technique [26, 27]; programming semantics [32]; and high level language implementations [4,6,29,30,31]. The perspective has been to view the high level languages as a class and to determine the set of linguistic constructs indicated. We now have these constructs in hand and may use them to develop a dynamic model for program execution.
Table II-1

Data modes

<table>
<thead>
<tr>
<th>mode</th>
<th>representation set</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>bool</td>
<td>{true, false}</td>
<td>false</td>
</tr>
<tr>
<td>char</td>
<td>{c: c ∈ Alphabet}</td>
<td>blank</td>
</tr>
<tr>
<td>mode</td>
<td>{m: m ∈ Modenames}</td>
<td>NULL</td>
</tr>
<tr>
<td>user</td>
<td>Ordered tabulation</td>
<td>zeroth item</td>
</tr>
<tr>
<td>int</td>
<td>{i: i ∈ Integers and</td>
<td>i</td>
</tr>
<tr>
<td>real</td>
<td>{r: r=x ∙ b^y where</td>
<td>x</td>
</tr>
<tr>
<td>cmpx</td>
<td>{z: z defined similar to real for x_r, x_i, y_r, y_i, etc}</td>
<td>0</td>
</tr>
</tbody>
</table>

Modenames

NULL  BOOL
CHAR  MODE
USER  INT
REAL  CMPX
PROC  ROW
STRUC REF

Any particular representation alphabet may be used for character variables. For arithmetic M, N, and b are implementation parameters.
Figure II-1

Value of mode row

<table>
<thead>
<tr>
<th>row</th>
<th>indir</th>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>final</td>
<td></td>
</tr>
</tbody>
</table>

Indir gives the mode of the sequence of values referenced by the row descriptor. Base is the location of the values. Initial and final denote the subscript bounds.
<table>
<thead>
<tr>
<th>Operators</th>
<th>Table II-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference*</td>
<td>* denotes multiple categories</td>
</tr>
<tr>
<td>Transput*</td>
<td>Reference</td>
</tr>
<tr>
<td>Rowing*</td>
<td>Transput</td>
</tr>
<tr>
<td>Calculators*</td>
<td>Take</td>
</tr>
<tr>
<td>Assign</td>
<td>Heap</td>
</tr>
<tr>
<td>Call</td>
<td>Free</td>
</tr>
<tr>
<td></td>
<td>Locate</td>
</tr>
<tr>
<td></td>
<td>Valuate</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>Logic</td>
</tr>
<tr>
<td></td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>Not</td>
</tr>
<tr>
<td></td>
<td>And</td>
</tr>
<tr>
<td></td>
<td>Calculators</td>
</tr>
<tr>
<td></td>
<td>Or</td>
</tr>
<tr>
<td></td>
<td>Xor</td>
</tr>
<tr>
<td></td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>Arithmetic</td>
</tr>
<tr>
<td></td>
<td>Arithmetic</td>
</tr>
<tr>
<td></td>
<td>Add</td>
</tr>
<tr>
<td></td>
<td>Subtract</td>
</tr>
<tr>
<td></td>
<td>Multiply</td>
</tr>
<tr>
<td></td>
<td>Divide</td>
</tr>
<tr>
<td></td>
<td>Mod</td>
</tr>
<tr>
<td></td>
<td>Exponential</td>
</tr>
<tr>
<td></td>
<td>Norm</td>
</tr>
<tr>
<td></td>
<td>Conjugate</td>
</tr>
<tr>
<td></td>
<td>h</td>
</tr>
</tbody>
</table>
### Table II-3

**Calculator Semantics**

<table>
<thead>
<tr>
<th>Subclass</th>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic</td>
<td>$\text{bool} \times \text{bool}$</td>
<td>$\text{bool}$</td>
</tr>
<tr>
<td>Precedence</td>
<td>$\text{ord} \times \text{ord}$</td>
<td>$\text{bool}$</td>
</tr>
<tr>
<td>Conformity</td>
<td>$\text{any} \times \text{any}$</td>
<td>$\text{bool}$</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>$\text{arit} \times \text{arit}$</td>
<td>$\text{arit}$</td>
</tr>
<tr>
<td>Identity</td>
<td>$\text{ref} \times \text{ref}$</td>
<td>$\text{bool}$</td>
</tr>
<tr>
<td>Translate</td>
<td>$\text{repr} \times \text{mode}$</td>
<td>$\text{repr}$</td>
</tr>
</tbody>
</table>

**Ordered modes (ord)**

- bool
- char
- user
- int
- real
- cmpx

**Arithmetic modes (arit)**

- int
- real
- cmpx

**Representation modes (repr)**

- row of bool
- char
- user
- int
- e

**Reference modes (ref)**

- name
- loc
- d
### Figure II-2

**Statements**

<table>
<thead>
<tr>
<th>Kind</th>
<th>Number of arguments</th>
<th>Argument types</th>
</tr>
</thead>
<tbody>
<tr>
<td>While</td>
<td>2</td>
<td>Both arguments are clauses.</td>
</tr>
<tr>
<td>Repeat</td>
<td>2</td>
<td>Both arguments are clauses.</td>
</tr>
<tr>
<td>If</td>
<td>3</td>
<td>All arguments are clauses.</td>
</tr>
<tr>
<td>Case</td>
<td>$N + 3$</td>
<td>The first argument is a literal positive integer, N. All others are clauses.</td>
</tr>
<tr>
<td>Sequence</td>
<td>$N + 1$</td>
<td>The first argument is a literal positive integer, N. All others are clauses.</td>
</tr>
<tr>
<td>Exit</td>
<td>1</td>
<td>The argument is an operand.</td>
</tr>
</tbody>
</table>
Chapter III

A Dynamic Process Model

In this chapter a dynamic process model is presented based on the linguistic constructs developed in Chapter II. This represents the second step in the specification of a language directed machine; that is, its translation from a linguistic specification to a mathematical or computational specification. In the development of linguistic primitives it was sufficient to consider task formulation as the specification of a solution to a problem. At this point we wish to consider the question of task evaluation; that is, the transformation which is applied to a particular task formulation to yield its solution. Such a transformation is a process. A process is considered to exist from time t-start to t-finish and to occupy during that interval a processor. Thus at any time t during the existence of a process it is composed of a) an algorithm or program specification, b) an execution record, which is composed of the universe of discourse and a locus of control, and c) a process indicator which denotes the current
processor state. Correspondingly a dynamic process model is presented which a) preserves program structure, b) maintains environments of access and loci of control, and c) provides for state transformations. The model presented is the outgrowth of work on tagged architecture of data type management and multilevel descriptor addressing for environment management [17,21]. Influential in determining the model design were Johnson's Contour Tree Model [18,19, 20]; Wells' model for MADCAP VI [33,34]; and the PAL blackboard evaluator [35]. Since it is intended as a basis for a register transfer implementation, the model and its evaluation is developed at a detailed syntactic and semantic level. Like the process it represents the model consists of an algorithm, an execution record, and a process indicator. These are presented in turn.

We first consider the algorithm. This is the representation of the source program as the formulation of the solution to a task. An algorithm is represented as a tree of statements and formulae as developed in Chapter II with formulae or exit statements at the leaf nodes. A statement is represented as an instruction and its arguments which are the subclauses of that statement. An instruction consists of a statement kind and a reference to a sequence
of clauses. Each of the arguments is either an instruction or a formula denoting a third statement. A formula is represented by an instruction with a kind field denoting formula and a reference to a sequence of objects. The object sequence consists of a literal denoting the number of objects in the sequence followed by the objects in order. An object is either an operand or an operator. The operators are as given in Chapter II. Each operand is either a statement instruction or an element of the universe of discourse which consists of a mode and a representation. All reference operands are of mode name (there are no locs in the static program algorithm since they are dynamically created by the process) and all procedure denotations have constant evaluation environment values as discussed in Chapter II. Thus a program is denoted by the instruction belonging to the root clause of the tree.

Figure III-2 gives the model's algorithmic representation of the Algol-like program shown in Figure III-1. This example is a formulation of the solution to finding four mutually nonattacking queens on a four by four grid. This formulation will be used in further examples. Note that a null clause (as in the case of an if clause with no else branch) is represented by a null instruction where it
appears as an argument. See, for example, the if clause at line J. The print names of variables are given in parentheses opposite the formula in which their allocation occurs. Note also that the allocation of multiple values requires repeated use of the take operator. This is because each invocation of the operator handles one level of a multi-level structure. Thus we have arrived at a concise description of an algorithm to which a formal evaluation mechanism may be applied.

The second component we wish to consider is that which models the execution record of a process. As opposed to the static algorithm, the execution record is a dynamic structure which is the result of program evaluation. This component is composed of the locus of control and the universe of discourse. The locus of control is modeled as a stack of instructions for clause and formula evaluation. The instructions which appear on the stack and their arguments are given in figure III-3. Thus the locus of control gives the set of statements and formulas within whose range the current point of process control is located. The universe of discourse is modeled as the set of constants known within the language definition plus the set of named variables accessible from the current point of control.
As noted in Chapter II, there are two sets of variables, contextual and dynamic. Contextual variables are modeled as a linked stack of environments within the program workspace. Each environment contains the names defined on that level. Each environment is allocated in the first free space after its caller and is linked to its containing environment. A typical environment stack configuration is shown in Figure III-4. Access link of the outer most environment refers to system. Note that the variables and the return link is null. Dynamic variables are modeled by a heap and an associative lexicon which is called the namelist. Multiple values (rows of values) and retained environments (see next paragraph) are allocated on the heap. The namelist consists of a name field and a value field. The location of a variable is its index in the namelist. A new occurrence of a variable is allocated in the first free location beyond the last occurrence of the same name. The last occurrence of a name is referenced or freed. Thus we see that the execution record is composed of a locus of control, a linked stack of environments, and a heap in a directly accessed memory, and a namelist in an associatively accessed memory.

At this point procedure and reference and row variables
bring about the question of reference management. Several strategies are possible. For the implementation lifetime well stacked (Berry et al [36], etc.) languages it is sufficient to prohibit uplevel and cross assignment of reference variables or uplevel assignment of procedure variables or their use in the heap. Alternatively uplevel assignment may tag an environment or variable in the stack and it may be allocated on the heap at block exit by a trap to a system routine where it is subject to garbage collection techniques. This strategy allows retained environments and variables without making the constructs which must be represented in hardware prohibitively complicated.

The third component of the process model is the process indicator. This component represents the current state of the process during evaluation. It is upon this component that the evaluation mechanism acts to provide task solution. It consists of the current point of control, the accessible universe, and the state value of the process. The point of control is the object or instruction within the immediately containing formula or statement respectively. In addition it may also consist of parameters peculiar to a particular statement, for example the number of statements
in a sequence. The accessible universe is the environment within which the current point of control is operative, along with its containing environments, as well as each of most recently allocated dynamic variable of a given name. Since each statement of formula returns a value, the state value of the process is the value of the statement or formula most recently evaluated. Thus we have described the semantic constructs on which the evaluation mechanism operates.

Process evaluation may be divided in two parts, formula evaluation and statement evaluation. Formula evaluation has been partially described in Chapter II. One further structure is needed for formula evaluation at this point. This is the working stack where the operands of a formula are retained until operated upon. Since statement and formula evaluation occur alternately, the operand stack is interlaced with the locus of control. Hence the need for an additional argument for a stacked formula (see Figure III-3). The evaluation of a formula causes its operands and operators to be stacked and executed in order. When an operator is encountered the arguments are removed from the stack and replaced by the resulting value. At the termination of the evaluation the resulting value is
taken as the value of the formula.

The evaluation of a statement on the other hand may require the evaluation of several subordinate clauses (statements or formulae) before evaluation is complete. Also, the operand of a formula may be a statement which is evaluated for the value it returns. Thus in clause evaluation we see that there are several semantic classes of action which are possible. First the evaluation of a statement or formula may be begun. Next, a statement (formula) may be suspended pending the evaluation of a subordinate clause (statement). The evaluation of a suspended statement (formula) may be continued when a subordinate clause (statement) terminates. Finally, either a statement or a formula may terminate. The semantics of evaluation for each clause type is shown in Table III-1. Just as the operand stack is used for stacking in partially evaluated formulae, the locus of control is used for stacking instructions (statements and formulae) which are to be continued. When a statement terminates it returns a value as given in Table III-1. When the root statement of the program tree terminates the process is completed. At this point the locus of control is empty.

The preceding paragraphs have presented a dynamic
process model based on the linguistic constructs developed in Chapter II. Although the process described is dynamic, only a static exposition has been presented this far. To alleviate this we again refer to the example of the queens program given in Figure III-1. In order to illustrate the dynamic execution of the process several snapshots of the execution record and process indicator are shown with a commentary on evaluation. Each snapshot consists of the locus of control, the universe of discourse and the state value at a particular point of evaluation. The point of control, that is the innermost clause of the locus of control is shown as the current instruction. It is this statement or formula which is currently being evaluated.

Figure III-5.a shows the execution record immediately after program entry, but prior to the evaluation of any clauses. The locus of control is initialized to a null instruction which will provide for program termination. The universe of discourse is initialized to an as yet empty environment which is contextually contained directly with the system environment, hence the access link refers to the system environment and the return link is null. Since evaluation has not commenced the state value is also null. The current instruction is the root of the program tree;
that is, a reference to a sequence of clauses at line A of Figure III-2.

Figure III-5.b shows the execution record during the evaluation of the formula at line B of Figure III-2 (allocation of rank). Its containing statement, the sequence at line A, has been stacked in the locus of control giving the number of clauses remaining in the sequence (3) and the location of the next clause (A+2). The formula at line B is shown partially evaluated. The operands at B+1 and B+2 have been stacked as temporary values. Next the take operator at B+3 will be applied to the mode: INT value on top of the temporary stack and a new variable will be allocated in the present environment and its location will be placed on the temporary stack. Finally, the assign operator at B+4 will give the newly allocated variable a value of int: 1. At this point the formula terminates with the value int: 1. Although not shown in the snapshot it is obviously necessary to maintain the number of objects remaining to be used in the current formula.

Figure III-5.c shows the execution record immediately prior to the formula from which the procedure Setqueen is called. Since this formula is the last clause of the sequence
of line A, the locus of control again contains only the null instruction. That is, the termination of this formula will also cause program termination. At this point a number of new variables have been allocated in the current environment and it is no longer empty.

Figure III-5.d shows the execution record after the procedure call. Both the return and access link of the new environment have been linked to the outermost environment. The integer variable file has been allocated and initialized to the value 1. The body of the repeat at line I and the predicate of the if contained within it have been entered. The current point of control is within the formula which is the predicate of the if. The and operator at K+15 has just been executed yielding a value of bool: true. Since this is the last object of the formula, the formula will terminate with this value and the if statement continued, selecting the first argument (then branch). Note the formula saved because of the procedure call. If the operand stack had been non-null it would have been saved immediately after the null instruction labeled z.

Figure III-5.e shows the execution record upon the finding of a solution. Note that the access links of each
call on the procedure Setqueen references the contextually containing environment (the main program), while the return links reference the calling environment. The point of control is the formula which is the predicate of the if at line N. Since rank (=5) is greater than four the value of the formula is \texttt{bool}: true indicating that a solution has been found. The formula terminates with this value and the if statement is continued, selecting the then branch which causes writing of the solution to the solution to the output.

In this chapter we have developed a semantic program model of process evaluation based on the linguistic constructs developed in Chapter II. This model is based on the evaluation of a program algorithm, modeled as a tree of statements and formulae, within a universe of discourse, modeled as a set of linked environments for contextually defined variables, and an associative list of names and values, a heap storage area for dynamically defined. A locus of control is developed to model the algorithmic behavior of the program. A process indicator is defined to model the current point of process evaluation, and the semantics of the evaluation mechanism is developed. Finally, a simple example is provided to illustrate the rudimentary
workings of the model. In the following chapter this model will be used to serve as the blueprint for the design of a language directed computer architecture.
program fourqueens=
    begin
        procedure setqueen=lambda: begin
            int file:=1;
            repeat
                if filefree[file] and diagfree[rank-file]
                    and skewfree[rank+file]
                    then
                        skewfree[rank+file]:= diagfree[rank-file]
                        := filefree[queen[rank]:=file]:=false;
                        if (rank:=rank+1) ≤ 4 then
                            setqueen()
                        else write(queen) fi;
                        filefree[file]:= diagfree[rank-file]:=
                            skewfree[file+(rank:=rank-1)]:=true
                    fi
                until (file:=file+1) > 4 stop
            end setqueen;
            int rank:=1;
            [1:4] int queen ;
            [2:8] bool skewfree:= [-3,3] bool diagfree:=
                [1:4] bool filefree:= true;
            setqueen()
        end fourqueens.

Note that the source of the assign operator is evaluated first as is the subscript of the index operator. All other evaluation is left to right as it appears in the program.
Figure III-2

l:  sequence: A;
B:  4, int: 1, mode: INT, take, :=. (rank)
C:  5, mode: INT, int: 4, mode: ROW, take, take; (queen)
D:  21, bool: true, mode: BOOL, int: 4, mode: ROW,
    take, take, :=, mode: BOOL, int:-3, int: 3, (filefree)
    mode: ROW, take, take, :=, mode: BOOL,           (diagfree)
    int: 2, int: 8, mode: ROW, take, take, :=       (skewfree)
E:  2, proc (1,F), call.
F:  0, sequence: G
G:  2, formula: H, repeat: I
H:  4, int: 1, mode: INT, take, :=                (file)
I:  if: J, formula: T
J:  formula: K, sequence: L, null
K:  15, name: file, name: filefree, index, name: rank,
    name: file, -, name: diagfree, index, and, name: rank,
    name: file, +, name: skewfree, index, and
L:  3, formula M, if: N, formula: S
M:  21, bool: false, name: file, name: rank, name: queen,
    index, :=, name: filefree, index, :=, name: rank,
    name: file, -, name: diagfree, index, :=, name: rank,
    name: file, +, name: skewfree, index, :=
P:  7, name: rank, int: 1, +, name: rank, :=, int: 4, ≤
Q:  2, proc: (1,F), call
R:  3, name: queen, evaluate, write
S:  21, bool: true, name: rank, int: 1, -, name: rank, :=
    name: file, +, name: skewfree, index, :=, name: rank,
    name: file, -, name: diagfree, index, :=, name: file,
    name: filefree, :=
T:  7, name: file, int: 1, +, name: file, :=, int: 4, >

Letters in the left margin denote the location of the first
in a sequence of items. The print names of variables are
placed in the right margin adjacent to the formulas which
cause their allocation.
Table III-1

clause type  semantics

repeat:  On entry a repeat symbol and a reference to
the predicate of the clause is stacked in
the locus of control and the body of the
clause is evaluated.

On occurrence of a repeat symbol in the locus
of control an until symbol and a reference
to the body of the clause are stacked and
the predicate of the clause is evaluated.

On occurrence of an until symbol in the locus
of control stack the state value must be
bool. If it is false then a repeat symbol
and a reference to the predicate of the
clause is stacked and the body is evaluated.
If it is true then evaluation of the next
item in the locus of control is continued.

while: On entry or the occurrence of a do symbol in
the locus of control a while symbol and a
reference to the body of the clause is stacked
and the predicate is evaluated.

On occurrence of a while symbol in the locus of control the state value must be a bool. If it is true then a do symbol and a reference to the predicate is stacked and the body is evaluated. If it is false then the next item in the locus of control is continued.

<table>
<thead>
<tr>
<th>clause type</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>On entry a sequence symbol, the number of arguments, and a reference to the sequence of arguments is stacked. On occurrence of a sequence symbol in the locus of control if the number is greater than one the first argument is removed and the sequence symbol, a new number, and a reference to the remaining sequence of arguments is stacked. The first argument is evaluated.</td>
</tr>
<tr>
<td>case:</td>
<td>On entry a case symbol, the number of alternatives, and a reference to the arguments is stacked and the predicate is evaluated.</td>
</tr>
</tbody>
</table>
On occurrence of a case symbol in the locus of control the state value must be an `int`. If it is outside the range of numbers it is set to zero (the default value). The corresponding alternative is evaluated.

**if:**

On entry an if symbol and a reference to the arguments is stacked and the predicate is evaluated.

On occurrence of an if symbol in the locus of control the statevalue must ve a `bool`. If it is true the first argument is evaluated, otherwise the second argument is evaluated.

**clause type semantics**

**exit:**

On entry the argument of the clause must be a positive `int`. This number of clauses is removed from the locus of control and the next topmost remaining clause is continued.

**formula:**

On entry the formula is evaluated as given in Chapter II and the present chapter. If one of the objects of the formula is a statement or a call on a procedure, a formula symbol, the number of objects remaining to be evaluated
in the formula, a reference to the objects
and the operand stack are stacked on the
locus of control and the corresponding state-
ment or procedure body is evaluated. In the
case of procedure evaluation a tag is also
stacked to indicate entry into a new contextual
environment.
On occurrence of a formula symbol in the
locus of control the operand stack is restored,
and if the environment tag is present the
containing environment is made current. The
formula is then continued.
Figure III-3
Clauses Suspended in Control Locus

<table>
<thead>
<tr>
<th>kinds</th>
<th>arguments</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do</td>
<td>@predicate</td>
<td>Predicate is the first argument of the While statement.</td>
</tr>
<tr>
<td>While</td>
<td>@body</td>
<td>Body is the second argument of the While statement.</td>
</tr>
<tr>
<td>Until</td>
<td>@body</td>
<td>Body is the first argument of the Repeat statement.</td>
</tr>
<tr>
<td>Repeat</td>
<td>@predicate</td>
<td>Predicate is the second argument of the Repeat statement.</td>
</tr>
<tr>
<td>If</td>
<td>@args</td>
<td>Args is a sequence of two clauses denoting the then and else branch of the If statement.</td>
</tr>
<tr>
<td>Sequence</td>
<td>number</td>
<td>Number indicates the quantity of subclasses remaining to be evaluated in the sequence. First is the next such clause.</td>
</tr>
<tr>
<td></td>
<td>@first</td>
<td></td>
</tr>
<tr>
<td>Case</td>
<td>number</td>
<td>Number indicates the quantity of alternative clauses in the Case statement. Args is a sequence of number+1 clauses. If the selector is out of bounds the N+1st &quot;default&quot; clause is chosen.</td>
</tr>
<tr>
<td></td>
<td>@args</td>
<td></td>
</tr>
<tr>
<td>Formula</td>
<td>@objects</td>
<td>Objects is the sequence of operators and operands remaining to be evaluated. Number gives the quantity remaining. Next is the most immediately containing statement of the formula. This allows the Control Locus and expression stack to share memory.</td>
</tr>
<tr>
<td></td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@next</td>
<td></td>
</tr>
</tbody>
</table>

Note: @ denotes "location of"
Figure III-4

Single environment

A particular universe of discourse consisting if a containing environment and two subsequently allocated environments textually contained directly within the first.
Figure III-5

| Sequence: A | system null |
| instruction | universe |
| null | null |
| locus | value |

\( a \)

| Formula: B+3 | system null |
| instruction | universe |
| Sequence: A+2 | mode: INT |
| 3 | int: 1 |
| null | value |

\( b \)

| Formula: E | instruction |
| null | bool: true |
| locus | Value |

| skewfree: | row of bool: d |
| 2 | 8 |
| d: | bool: true |
| bool: true |
| bool: true |
| bool: true |
| bool: true |
| bool: true |
| bool: true |

| diagfree: | row of bool: c |
| -3 | 3 |
| c: | bool: true |
| bool: true |
| bool: true |
| bool: true |
| bool: true |
| bool: true |
| bool: true |

| filefree: | row of bool: b |
| 1 | 4 |
| b: | bool: true |
| bool: true |
| bool: true |
| bool: true |

| queen: | row of int: a |
| 1 | 4 |
| a: | int: 0 |
| int: 0 |
| int: 0 |
| int: 0 |

| rank: | bool: true |
| system null |
| universe |
file:

<table>
<thead>
<tr>
<th>skewfree:</th>
<th>row of bool: d</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d;</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>diagfree:</th>
<th>row of bool: c</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c;</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>filefree:</th>
<th>row of bool: b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b:</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
<tr>
<td>bool: true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>queen:</th>
<th>row of int: a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int: 1</td>
</tr>
<tr>
<td>int: 0</td>
</tr>
<tr>
<td>int: 0</td>
</tr>
<tr>
<td>int: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rank:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int: 1</td>
</tr>
</tbody>
</table>

| system null |
CHAPTER IV

A Computer Architecture

In the first chapter an argument is made for a process of computing machine design analogous to program refinement. We further note that the steps in the refinement of a language directed machine are the development of linguistic primitives, the specification of a process model, and the design of a machine architecture. The first two steps have been discussed in Chapters II and III. The present chapter is concerned with the final step in machine design; that is, the use of the process model as an architectural blueprint for the actual machine design. The language directed computer will be described at a register transfer level. A set of process registers for program evaluation and a control mechanism based on the registers' contents will be described. Field lengths and address calculations are not given as these depend on addressing bandwidth (say byte or word of n bits). Logically an address (location) is a relative offset in a bounded segment. It is modeled at this level.
The algorithm and process workspace (control locus and set of active environments) are modeled as register addressed memory segments. The algorithm is read only. The set of dynamic variables is modeled by a register addressed associative memory and a register addressed heap memory. Portions of heap memory are read protected, depending on whether they are free or allocated. Heap management instructions cause traps to supervisory routines (or microcode). The set of registers required for formula and clause evaluation will be presented and the control algorithms which operate on the contents of these registers will be described. The relation of each control algorithms to the structures shown in the snapshots of Chapter III will be described.

Figure IV-1 gives a list of the processor registers. The registers may be divided into groups. The first is the set of memory descriptor registers, each of which references a given segment for the duration of the process. They are as follows: ALGO: This is a descriptor to a static segment which contains the representation of the program algorithm. Read reference may be made to this segment for statements, formulae, operands, and operands. WORK: This is a descriptor to the locus of control and the
set of contextual environments in the universe of discourse. Reference may be made via the locus of control for the stacking or unstacking of statements or formulae, or via the current environment base for the creation, binding or deletion of environments or variables.

NAMELIST: This is a descriptor to an associatively addressed table which contains the name and the value of each dynamically allocated variable. The table may also be addressed sequentially and a variable's location is its offset in the table. This allows the full implementation of the locate operator.

HEAP: This is a descriptor to a dynamic storage area where multiple values are kept. Heap management operations cause traps to system software (or microcode). The preceding set of registers is sufficient to model memory usage for the process. The BCPL implementation models only the lifetime well stacked subset of the model, hence uses only the ALGO and WORK memory registers. However, in describing the evaluation mechanism the details of both contextual and dynamic allocation will be considered.

The second group is the process state registers. The evaluation mechanism is based upon the contents of these registers. They are introduced in the order of their
appearance in program evaluation. At the clause evaluation level, instructions whose evaluation is to be commenced are placed in ENTRY. If an instruction is to be suspended it is placed in the Control Locus, an instruction stack in the segment referenced by WORK. The address of the stack top is maintained in CONTROLPTR. Statements which are continued are loaded in INSTR. The evaluation of sequence and case statements and of formulae requires the use of a literal integer, the NUMBER register serves this purpose. At this point we have enumerated the registers required by statement evaluation except for its value as an expression. Thus we include TEMP, which is in fact the current working value of the process. Thus we may turn now to formula evaluation.

An object of a formula is loaded into OPBUS. At this point TEMP is logically the top element of an Operand Stack. The remaining elements are stacked at the top of the current Control Locus. The top of the Operand Stack is referenced by OPERANDPTR. If the object in OPBUS is an element of the universe of discourse it is placed in TEMP pushing the Operand Stack. If it is an operator, then its arguments from the top of the Operand Stack are replaced by the result, which becomes the current value of TEMP.
Finally, if it is a statement, the formula is suspended and the statement is placed in ENTRY.

What is left then is to provide registers for the various operators. SEND is used as a general purpose secondary operand register for binary operations and may be loaded only from TEMP. Referencing operators require the use of ENVBASE, which references the base of the current environment and NAMEPTR, which references the next free location in the current environment. BASE is used for segment offset calculations in allocation and call. UPPER and LOWER are used for subscript bounds in rowing operators. Figure IV-1 gives a summary of the registers used.

The preceding paragraphs have served two purposes. First, to denote the registers used in designing a computer architecture from a dynamic process model. Second, to indicate their usage in a broad semantic outline of the evaluation mechanism. We turn now to a more complete description of that evaluation mechanism. The evaluation mechanism will be described as a set of control logarithms which operate on the contents of the process registers. These will be developed in the order of the above outline.

Since program evaluation is essentially clause evaluation
in an empty environment we may begin with clause evaluation. The following is the control algorithm for clause evaluation. The symbols *:= and :=* denote assignment to and from a stack respectively.

While ENTRY is not null do

   Enter new clause;

   while ENTRY is null & Control Locus is not null do

      Continue old clause

   od

od.

Given a well formed program tree as input the preceding algorithm produces a treewalk evaluation. What remains is to define the subordinate functions. We consider first entry to clause evaluation:

Case kind of ENTRY is

Exit: Unstack Control Locus;

   Entry:= null; return.

Formula: NUMBER:= ALGO[ENTRY]; ENTRY+:=1;

   TEMP:= null; Evaluate formula; return.

Case:

Sequence: Control Locus*:=ALGO[ENTRY].

If:
While:

Do: exit case.

default: "Kind of ENTRY is not valid clause"

esac;

Control Locus* := ENTRY; ENTRY := ALGO[ENTRY]

The preceding algorithm provides a formal semantic definition of clause entry at a register transfer level. The task of unstacking the Control Locus remains to be defined. Note that errors in application of the linguistic constructs of Chapter II are detected directly at a hardware level, thus making it possible to verify during evaluation that the program algorithm is in fact a well formed tree of well formed formulae as defined in Chapters II and III. It could be assumed that a compiler exists which produced a correct tree, obviating the need for error semantics. However, from the viewpoint of machine design the compiler writer must be considered a user as much as the problem solver. Thus even his programs must be subject to the semantic protection which may be provided. This philosophy is maintained throughout in the choice of runtime diagnostics.

The second facet of clause evaluation we wish to examine is the continuation of a suspended clause. The algorithm follows:
INSTR:=* Control Locus; ENTRY:= null;

If kind of INSTR is Sequence or Case
then NUMBER:=* Control Locus fi;
if kind of INSTR is Case or While or Until
then while mode of TEMP is reference do valuate operator od fi;
case kind of INSTR is

Formula: OPERANDPTR:=@ Control Locus + 1;

CONTROLPTR:=* Control Locus;

if environment had changed then
(trap to save if retained;)

NAMEPTR:= ENVBASE;

ENVBASE:= returnptr of WORK[ENVBASE]
fi; ENTRY:= INSTR;

Evaluate Formula; return.

Case: if mode of TEMP is int
then ENTRY:= ALGO[ENTRY+TEMP]
else "mode of TEMP is invalid for Case statement" fi;
return.

Sequence: NUMBER:=:=l; ENTRY:= ALGO[location of INSTR];

INSTR+::=l;

if NUMBER l then Control Locus*:= NUMBER
else return fi.

If: if mode of TEMP is bool then if val of TEMP
then INSTR+:=1 fi;

ENTRY:= ALGO[INSTR]

else "mode of TEMP is invalid for If Statement" fi;

return.

While: if mode of TEMP is bool

then if val of TEMP

then pop Operand Stack (into TEMP; return

else Kind of INSTR:= Do; pop Operand Stack;

ENTRY:= ALGO[location of INSTR]; INSTR-:=1;

fi

else "mode of TEMP is invalid for While statement" fi.

Repeat: Operand Stack*:= null (i.e. TEMP is null);

kind of INSTR:= Until; ENTRY:= ALGO[location of INSTR];

INSTR-:=1.

Do: Operand Stack*:= null; kind of INSTR:= While;

ENTRY:= ALGO[INSTR]; INSTR+:=1.

Until: if mode of TEMP is bool

then if not val of TEMP

then pop Operand Stack; return

else kind of INSTR:= Repeat; pop Operand Stack;

ENTRY:= ALGO[location of INSTR];

INSTR+:=1

fi
else "mode of TEMP is invalid for Until Statement" fi.

default: "invalid clause kind in INSTR"

esac; Control Locus*: = INSTR.

The preceding algorithm provides for the continuation of clause evaluation as given in Figure III-3 and Table III-1. The three preceding control algorithms, taken as a whole effectively define clause evaluation. The first algorithm determines the order of clause entry and continuation to provide a proper walk of the program tree. The second algorithm initiates evaluation of those clauses which appear in the program algorithm segment and the third control algorithm continues evaluation of those clauses suspended in the Control Locus. Normal termination is implicitly defined by the second and third algorithms. With the description of Control Locus unstacking for multiple exit the description of statement evaluation will be complete. The algorithm follows:

OPBUS:= ALGO[location of ENTRY]; Operand Stack*: = OPBUS;

while mode of TEMP is reference mode do valuate operator od;
if mode of TEMP is int and val of TEMP = depth of Control Locus then NUMBER:= val of TEMP; pop Operand Stack;

    while NUMBER--:=1 0 do
INSTR:="* Control Locus;

case kind of INSTR is

  Sequence:

  Case: pop Control Locus.

  Formula: pop Control Locus;

     OPERANDPTR:= CONTROLPTR;

     CONTROLPTR:="* Control Locus;

     if environment had changed then

        (trap to save retained environment;)

     NAMEPTR:= ENVBASE;

     ENVBASE:= returnptr of WORK[ENVBASE]

     fi.

     If: While: Repeat:

     Do: Until: exit case.

     default: "invalid clause in control locus

               during exit statement"

     esac;

     od; ENTRY:= null

else "invalid operand of Exit statement" fi.

The preceeding control algorithm provides for the pre-
emptive termination of a suspended clause to any depth of
nesting via unstacking of the Control Locus. We have
considered then all of the control algorithms necessary
for clause evaluation and may turn now to formula evaluation.

We now consider the set of algorithms invoked by the directive in the second and third clause evaluation algorithms "Evaluate formula." We are first concerned with the recognition of the sequence of objects belonging to a formula and their evaluation. The initial formula evaluation algorithms follows:

While NUMBER:=1 0 do

    OPBUS:= ALGO[ENTRY]; ENTRY+=1;

    if OPBUS is in the universe of discourse then

        Operand Stack*:= OPBUS

        if mode of TEMP is proc then envir of TEMP:= ENVBASE;

            while level of TEMP:=1 0 do

                envir of TEMP:= access of WORK [envir of TEMP]

            od

    fi

    elseif OPBUS is a statement then

        Operand Stack*:= CONTROLPTR, null; (ie. TEMP=null)

        CONTROLPTR:= OPERANDPTR; Control Locus*:= NUMBER;

        Control Locus*:= ENTRY; ENTRY:= OPBUS: return

    elseif OPBUS is an operator then Apply operation

    else "invalid object of formula in OPBUS" fi
od;
if OPERANDPTR ≠ CONTROLPTR then "formula not well formed" fi.

We see that if the object of a formula is an operand in the universe of discourse it is stacked in working storage. If if is a statement to be evaluated then the formula is suspended and the statement instruction is placed in ENTRY. Finally, the object may be an operator, in which case the corresponding operation is performed. Note that when the operand stacked is a proc that the proper contextually containing environment is appended to its value. If the proc value is subsequently assigned outside the semantic range of its containing environment then the environment may be marked for retention at block exit.

To complete the description of formula evaluation only the application of the individual operators remain. The arguments of an operator are removed from the top of the Operand Stack and replaced by the result. Thus the domain of values for the primitive operators becomes the elements of the universe of discourse defined in Chapter II. We recall that at that point we introduced the reference modes name and loc as representations for naming and retaining in memory a particular element of the universe of discourse. We now deal with the question of substitution a value for
a reference in the course of an actual computation. To
do so we recall that if the argument of the valuate opera-
tor is a reference mode the result is the value referenced
(e.g. valuate (loc int: 3)= int: 3). Given this operator
we may now semantically model variable binding by the
following control algorithm when the argument of an opera-
tor must be an element of a nonreference mode;

While mode of TEMP is reference mode
Do TEMP:= valuate(TEMP) od.

This action will be denoted by the term dereferencing. In
the postfix formula evaluator this allows the maximum in
binding freedom, as bindings are delayed as far as possible
but earlier bindings may be obtained through the explicit
use of valuate and locate. Thus we could simply consider
operator application a case choice on the operation code
and give the result and side effects of each operation.

We will instead proceed categorically as presented in
Chapter II. (See Table II-2) The following control algo-
ритms are representations of operator and classes of
operators. The level of presentation will indicate what
values are found in and assigned to process registers, but
in many cases actual computation will not be elaborated.
In such cases the methods will be clearly indicated.

The first class we consider is referencing operators. We have already seen the effect of valuate. Similarly, locate binds the name of a variable to its current location. The take operator requires a variable number of arguments, its control algorithm follows:

Case mode of TEMP is

row: name:

loc: at WORK[NAMEPTR] allocate a row descriptor or reference variable whose value is TEMP; TEMP becomes loc ROW or loc REF addressing the allocated value.

mode:

case val of TEMP is

ROW: pop Operand Stack; dereference TEMP;

if mode of TEMP is int then

UPPER:= val of TEMP

else "invalid subscript mode in row allocation" fi;

pop Operand Stack; dereference TEMP;

if mode of TEMP is int then

LOWER:= val of TEMP;

pop Operand Stack; dereference TEMP
else LOWER:= 1 fi;

if mode of TEMP is row and is the most recently allocated element of the current environment then allocate at WORK[NAMEPTR] UPPER - LOWER additional copies of the structure described in TEMP; allocate a row of descriptors referencing the structures; TEMP becomes a row of ROW referencing the segment of descriptors

elsif mode of TEMP is mode then
allocate UPPER - LOWER + l items of the default value whose mode is val of TEMP;
TEMP becomes a row of mode to the sequence of elements

else "invalid mode of TEMP for row allocation" fi.

STRUCT: pop Operand Stack; dereference TEMP;
if mode of TEMP is int then UPPER:= val of TEMP else "invalid mode of TEMP in structure allocation" fi;
from 1 to UPPER do
pop Operand Stack; dereference;
if mode of TEMP is reference mode or row then allocate a descriptor or reference variable
89

which becomes TEMP

elsif mode of TEMP is mode then

allocate an item of that mode with the

the default value

else allocate an item whose mode and value is TEMP

fi;

od; TEMP becomes a row of MIXED mode descriptor to

the sequence of items allocated.

default: allocate a default value variable whose mode

is val of TEMP; TEMP becomes a loc to the variable

esac.

default: allocate an item initialized to TEMP;

TEMP becomes a loc referencing the value

esac.

The preceding control algorithm provides for the allocation

of contextual variables within the universe of discourse.

Simple variables may be taken initialized to a default

value or to a particular program value. Structures and

multiple arrays may be allocated by applying the operator
to items which are the result of the operator, thus allowing

for an inductive definition at the hardware level. For

example, the postfix polish formula: int: 0, int: -1, int:

1, mode: ROW, take, bool: true, take, int: 2, mode: STRUC,
take; allocates a **row** of MIXED mode to a sequence of two items which are a **bool** and a **row** of INT etc. At this point we have a complete description for the creation of variables in the current environment.

We next consider the allocation of dynamic variables. We may take the take algorithm as the model for heap allocation indicating the necessary changes. If the first argument of the operator is a **name** then the element allocated must be a simple value (data mode, reference mode, or **row** descriptor). The corresponding name and value placed in the first free location of the NAMELIST beyond the last occurrence of the same name. TEMP becomes the location of the variable. Otherwise a sequence of elements is taken in the region referenced by the HEAP register. The take algorithm is followed for multiple values, with the address calculations replaced by memory request traps to a storage exchange algorithm in software or microcode. TEMP becomes a **row** of mode descriptor to the sequence.

The final referencing operator is free. Any dynamic name location or segment may be freed. A service trap may occur at this point to delete dangling references. Contextual variables may be deleted in a last in first out order only. Thus if its argument is a contextual variable it must
be the most recently allocated item in the current environment. With the exposition of the referencing operators we have formalized the notion of a variable, including its creation and deletion, and the binding of a name or location to its corresponding value.

Since the assign operator may affect the binding of a variable to a particular value, assignment is considered next. Semantically the assign operator may be considered to replace a variable by a subsequent variable of the same name. Continuity conditions are usually enforced, say a particular variable name must maintain values in the same mode throughout its lifetime in a given context. Thus the common programming notion of a variable is transformed to into a sequential chain of "continuous" variables in the time domain of the program evaluation. Thus the application of a referencing operator prior or subsequent to a given assignment may produce different values or may be semantically meaningful only with a particular ordering of the operators. The control algorithm for the assign operator follows:

If mode of TEMP is name then locate(TEMP) fi;
SEND:= TEMP; pop Operand Stack;

    if mode of TEMP is reference mode then
case mode of SEND is

    loc: if mode of WORK[SEND] is reference mode
        then WORK[SEND]:= TEMP; return fi.

    row: if indir of SEND is REF
        then for i from 0 to UPPER-LOWER
            do WORK[location of SEND]:= TEMP od;
        return
    fi

esac; dereference TEMP

fi;

if mode of TEMP is proc or data mode then

    case mode of SEND is

        loc: if mode of WORK[SEND]:= mode of TEMP
            then WORK[SEND]:= TEMP
            else "mode disagreement in assign" fi.

        row: if indir of SEND = mode of TEMP
            then for i from 0 to UPPER-LOWER
                do WORK[SEND+i]:= TEMP od
            else "mode disagreement in assign" fi

esac

elsif mode of TEMP is row then

    if mode of SEND is loc and mode of WORK[SEND] is row
    then WORK[SEND]:= TEMP;

    row
elsif mode of SEND is row and subscripts match
then assign the elements descriptored by TEMP
item by item
else "mode disagreement in assign" fi
else "invalid source mode in assign" fi
else "invalid destination mode in assign" fi.

Scalar to scalar, vector to vector and vector expansion
of a scalar is provided by the assign operator. During
evaluation of the algorithm the following parallel action
is sufficient to implement a strategy of environment
management. When the destination address is calculated
it compared with the current environment base. If it is
outside the current environment or in the heap or namelist
and the source value contains a location the variable or
sequence at that location is marked for retention at block
exit. Names are bound to their corresponding locs on
uplevel assignment. The result of the operation is the
value of the source. Rows (and structures) are represented
as a sequence of items and a row descriptor to that sequence.
For row operands the sequence of values may appear either
in memory or along with the descriptor in the Operand Stack.
The following modification (restriction) must then be made
in assign: if either argument is on the Operand Stack
assignment must be item by item for row operands.

Since the exposition of the assign operator has dealt with rows of values, we turn next to the rowing operators. This is the first class of operators which as a group create no side effects, but return a value only. For this reason we may view them as primitive operators in the evaluation mechanism and state their semantics axiomatically. The slice operator requires a row of mode and a row of int as arguments. The elements of the first indexed by elements of the second are copied to the Operand Stack. TEMP becomes a row descriptor to the resulting sequence of values. The index operator allows multidimensional subscription of row values as indicated in Chapter II. If the element indexed is a row of mode on the Operand Stack, the portion subscripted is recopied and TEMP becomes a row descriptor to the values. Otherwise TEMP becomes a loc referencing the element indexed. The Third rowing operator is list. TEMP becomes a row descriptor to the sequence of values from the operand stack. The last three rowing operators are upper, lower and length. All are monadic operators which provide bound information and return an int value to TEMP.

At this point the only operator remaining which can
suspend formulae and create new environments is the call operator. It requires a variable number of arguments, the first of which must be a proc value of N parameters. The next N arguments of the operator are the actual parameters. Block entry may be modeled as a procedure of no parameters. The control algorithm follows.

 Dereference TEMP:

 if mode of TEMP is proc then

    returnptr of WORK[NAMEPTR] := ENVBASE;

    accessptr of WORK[NAMEPTR] := envir of TEMP;

    ENVBASE := NAMEPTR; NAMEPTR- := l;

    UPPER := ALGO[location of TEMP];

    BASE := ALGO[location of TEMP + 1];

    from l to UPPER do

       pop Operand Stack; apply take operator

       od;

    Operand Stack* := CONTROLLPTR, null;

    CONTROLLPTR := OPERANDPTR; Control Locus* := NUMBER;

    Control Locus := ENTRY; ENTRY := BASE

 else "TEMP invalid mode for call" fi.

In short a new offspring environment is created, parameters are passed, and the formula in which the procedure call
occurs is suspended. With the completion of the call operator all operators which deal with variable binding or environment manipulation have been described. The remaining operators are concerned with the transformation of values.

The next class of operators we consider is the calculators. Of these the identity operator is unique in that it does not operate on data modes. Identity test two reference values to see if they bind to the same location, the result is bool: true or false. Again calculators produce no side effects, but return a value only. Hence we may view them also as primitive elements of the evaluation mechanism and state their effect axiomatically rather than via a control algorithm. Thus TEMP becomes the result of the operator as given in Chapter II. Operands are checked for mode agreement and semantic errors in the use of values are detected. Calculators may be applied to vector or scalar operands. Thus when any calculator operator (other than translate, which takes scalars only) is applied the following control algorithm is used:

If operator is monadic then

   if mode of TEMP is row then

      do operation item by item;
TEMP:= row descriptor of result

else TEMP:= operator(TEMP)

elsif mode of TEMP is row then

    if mode of SEND is row

        then do operation item by item

    else do scalar vector expansion fi

elsif mode of SEND is row then

    do scalar vector expansion

else TEMP:= operator(TEMP, SEND) fi.

Thus we have considered a general class of calculator operators which may produce scalar or vector values.

The final class of operators we consider is the transput operators read and write. The read operator removes an element from the input queue and places it on the working stack. The write operator reproduces the top of the working stack on the output queue. A read operand must be a nonreference mode. A write operand may be any mode. References are given as environment, offset pairs.

With the exposition of the various classes of operators we have completed the description of the semantics for the high level computer architecture at a register gating level. The register set and the control algorithms given above provide a program evaluation model in terms of that
architecture. We now have sufficient information to construct a register gating diagram of the machine. This is given in Figure IV-2. The registers are divided into four functional groups according to their usage. The uses are statement evaluation, formula evaluation, allocation and environment manipulation, and operand and operator evaluation. A particular register may appear in more than one group as the WORK register, which denotes the program workspace appears in all four groups. Table IV-1 gives a list of the register gatings shown in Figure IV-2 and their corresponding control signals.

At this point the third step in the refinement, the design of the machine, is complete. A BCPL simulation of the register transfer machine was written and tested on the queen's problem of Chapter III in order to provide a comparison of the basic machine with more conventional design. The simulation implements the lifetime well-stacked subset of the process model, scalar calculations, and scalar and scalar to vector assignment. We have maintained in our hardware representation the linguistic primitives developed in Chapter II. In the following chapter we will examine the machine to determine its advantages and disadvantages with respect to conventional
### Figure IV-1

Memory Descriptor Registers

<table>
<thead>
<tr>
<th>name</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALGO</td>
<td>Program algorithm</td>
</tr>
<tr>
<td>WORK</td>
<td>Contextual variables and Control Locus</td>
</tr>
<tr>
<td>NAMELIST</td>
<td>Dynamic variable table (associative)</td>
</tr>
<tr>
<td>HEAP</td>
<td>Dynamic memory for multiple values</td>
</tr>
</tbody>
</table>

#### Processor State Registers

<table>
<thead>
<tr>
<th>name</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTRY</td>
<td>Clause to be entered</td>
</tr>
<tr>
<td>INSTR</td>
<td>Clause to be continued</td>
</tr>
<tr>
<td>CONTROLPTR</td>
<td>Control Locus</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Formula, Case statement, or Sequence statement</td>
</tr>
<tr>
<td>TEMP</td>
<td>Current state value</td>
</tr>
<tr>
<td>OPBUS</td>
<td>Objects of formulae</td>
</tr>
<tr>
<td>SEND</td>
<td>Secondary operand of operators</td>
</tr>
<tr>
<td>BASE</td>
<td>Address calculations</td>
</tr>
<tr>
<td>LOWER</td>
<td>Subscript calculations</td>
</tr>
<tr>
<td>UPPER</td>
<td>Subscript calculations</td>
</tr>
<tr>
<td>OPERANDPTR</td>
<td>Operand Stack</td>
</tr>
</tbody>
</table>


architectures. In particular, the data transfers required by the high level language machine for various constructs will be compared analytically with conventional machines, and program code used by simulator is compared with that produced by high level compilers on conventional machines.
Figure IV-2

Statement evaluation.

Formula evaluation.

Note: Double lines indicate data transfers, single lines are addresses for fetches or stores from or to memory segments. Numerals refer to control signals in table IV-1.
Allocation and environment management.

Operand and operator evaluation.
### Table IV-1

Register Gating

<table>
<thead>
<tr>
<th>No.</th>
<th>Gate from</th>
<th>Gate to</th>
<th>Control condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ALGO[ENTRY]</td>
<td>NUMBER</td>
<td>Formula entry</td>
</tr>
<tr>
<td>2</td>
<td>ALGO[ENTRY+TEMP]</td>
<td>ENTRY</td>
<td>Case statement choice</td>
</tr>
<tr>
<td>3</td>
<td>ALGO[ENTRY]</td>
<td>ENTRY</td>
<td>Clause entry</td>
</tr>
<tr>
<td>4</td>
<td>ALGO[ENTRY]</td>
<td>WORK[CONTROLPTR]</td>
<td>Entry to case or seq.</td>
</tr>
<tr>
<td>5</td>
<td>WORK[CONTROLPTR]</td>
<td>NUMBER</td>
<td>Cont. case, seq., form.</td>
</tr>
<tr>
<td>6</td>
<td>NUMBER</td>
<td>WORK[CONTROLPTR]</td>
<td>Suspend formula</td>
</tr>
<tr>
<td>7</td>
<td>WORK[CONTROLPTR]</td>
<td>CONTROLPTR</td>
<td>Continue formula</td>
</tr>
<tr>
<td>8</td>
<td>WORK[CONTROLPTR]</td>
<td>INSTR</td>
<td>Continue any clause</td>
</tr>
<tr>
<td>9</td>
<td>INSTR</td>
<td>WORK[CONTROLPTR]</td>
<td>Suspend continued clause</td>
</tr>
<tr>
<td>10</td>
<td>ENTRY</td>
<td>WORK[CONTROLPTR]</td>
<td>Suspend entered clause</td>
</tr>
</tbody>
</table>

| 11  | OPBUS         | TEMP            | Stack operand                      |
| 12  | OPBUS         | ENTRY           | Eval. statement opnd.              |
| 13  | WORK[OPERANDPTR] | TEMP         | Unstack operand                    |
| 14  | TEMP          | WORK[OPERANDPTR]| Stack operand                      |
| 15  | CONTROLPTR    | OPERANDPTR      | Init. operand stack                |
| 16  | CONTROLPTR    | WORK[OPERANDPTR]| Suspend formula                    |
| 17  | ALGO[ENTRY]   | OPBUS           | Get formula object                 |

b

Note: Fetches and stores to locations in memory segments are denoted by subscription.
<table>
<thead>
<tr>
<th>No.</th>
<th>Gate from</th>
<th>Gate to</th>
<th>Control Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>ENVBASE</td>
<td>NAMEPTR</td>
<td>Restore return envir.</td>
</tr>
<tr>
<td>19</td>
<td>NAMEPTR</td>
<td>ENVBASE</td>
<td>Create new environment</td>
</tr>
<tr>
<td>20</td>
<td>BASE</td>
<td>TEMP</td>
<td>Row descriptor calc.</td>
</tr>
<tr>
<td>21</td>
<td>NAMEPTR+BASE</td>
<td>TEMP</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>NAMEPTR</td>
<td>BASE</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>ENVBASE</td>
<td>WORK [NAMEPTR]</td>
<td>Create new environment</td>
</tr>
<tr>
<td>24</td>
<td>ENVBASE</td>
<td>TEMP</td>
<td>Stack proc; dereference</td>
</tr>
<tr>
<td>25</td>
<td>TEMP</td>
<td>WORK [NAMEPTR]</td>
<td>Create envir; alloc. var.</td>
</tr>
<tr>
<td>26</td>
<td>WORK [ENVBASE]</td>
<td>ENVBASE</td>
<td>Restore return envir.</td>
</tr>
<tr>
<td>27</td>
<td>UPPER</td>
<td>TEMP</td>
<td>Row descriptor calc.</td>
</tr>
<tr>
<td>28</td>
<td>TEMP</td>
<td>UPPER</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>TEMP</td>
<td>LOWER</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>LOWER</td>
<td>TEMP</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>NAMEPTR</td>
<td>TEMP</td>
<td>Create loc value</td>
</tr>
</tbody>
</table>

Operand and operator evaluation.

<table>
<thead>
<tr>
<th>No.</th>
<th>Gate from</th>
<th>Gate to</th>
<th>Control Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>TEMP</td>
<td>WORK [SEND]</td>
<td>Assignment</td>
</tr>
<tr>
<td>33</td>
<td>TEMP</td>
<td>WORK [TEMP]</td>
<td>Dereference</td>
</tr>
<tr>
<td>34</td>
<td>TEMP</td>
<td>SEND</td>
<td>Any dyadic operator</td>
</tr>
<tr>
<td>35</td>
<td>ALGO [TEMP]</td>
<td>ENTRY</td>
<td>Call operator</td>
</tr>
<tr>
<td>36</td>
<td>SEND+LOWER+TEMP</td>
<td>TEMP</td>
<td>Index and slice</td>
</tr>
</tbody>
</table>
CHAPTER V

An Evaluation and Summary

This thesis has outlined the stepwise refinement of a high level language directed computer architecture. A set of linguistic primitives were developed which were used for the construction of a dynamic process mode. This model was in turn used as the architectural blueprint for a register transfer level machine. Chapters II, III, and IV have provided the exposition of each of these areas respectively. We now wish to examine the machine produced to determine its qualifications as a language directed machine architecture. For this purpose the chapter is divided into two areas. We will first examine the goals developed in Chapter I and observe how the structural integrity of the machine serves these goals. Secondly, the relationship between the machine architecture and high level programming linguistics will be examined. The limitations of the current implementation will be discussed and directions for further refinement will be indicated.

We recall from Chapter I that by considering the
nature of a computation as an abstract process we were able to formulate several goals regarding the improvement of high level computation systems over conventional techniques. They are:

1. Provide more powerful linguistic constructs at the machine level
2. Reduce compiler complexity.
4. Reduce interactive complexity.

Each of these areas will be considered and the relevant machine constructs examined.

We wish to consider the semantic power of the linguistic constructs modeled. We first consider the division of the universe of discourse into classes of values by the representation of modes (data types). Those modes designated as data modes (primitive types) provide the basic set of elements from which the universe of discourse is constructed. Mode row provides aggregations of values. Name and loc allow variable binding and complex data structures. In addition to the obvious function of classifying values hardware representation of modes provide the following benefits:

1) dereferencing. The hardware indirection that
occurs when a primitive operator is applied to a reference
mode operand represents a savings of one instruction and
one memory fetch for each level of indirection over machines
in which tagging for indirection is not employed. Consider
the expression A+5 within the scope of the declaration REF
INT A; then an indirection on A must either be explicitly
provided by the programmer or generated by the compiler.

2) structures. The use of rows of mixed values and
modes name and loc permit the construction of arbitrarily
complex data structures.

3) binding. The semantic distinction between the
names and locations of variables as references permits
considerable freedom in binding these to their respective
values.

4) arithmetic. The use of modes real, int, and
cmpx allows the universal application of arithmetic opera-
tors to the fields of integers, reals, and complexes. Where
the integers are contained in the reals are contained in
the complexes and where the mode of the result is the union
(supremum) of the modes of the operands.

5) user mode. The use of user defined (enumerated)
data modes in case statements and subscripting allows a
programming abstraction recommended by Hoare in Structured
Programming [22].

6) modenames. The use of modenames as values of mode \texttt{mode} and and the allocate and conformity operators permits the construction of compilers at an advanced semantic level with little or no software service required for variable manipulation or mode determination.

In addition, the use of formulae and statements as basic machine instructions provides two related benefits. First, the syntactic structure of the program is directly represented in hardware. The most arbitrary decision was made in the hardware representation of formulae. For full semantic generality in variable binding and procedure closure it would be necessary to allow the ordering of the evaluation of a formula at run time, however the semantic complexity of the evaluation mechanism would be far greater than that formula evaluation with the ordering determined at compile time. Since formula ordering must then be done at compile time, postfix notation is chosen as the machine representation for several reasons. First, it presents a standard form to which various parenthesized forms may be easily translated. Second, it permits delaying variable binding until operator application, thereby permitting the maximum binding freedom within a static representation.
Further, variations in binding strategies can be achieved through explicit use of the locate and valuate operators.

Statement instructions provide the selection, iteration, sequencing and range exit activities presented in Chapter II. For this reason, the set of possible running programs that are recognized by the hardware is semantically restricted to those which can be constructed using the techniques of stepwise refinement. Hence a program may not represent an arbitrary flow graph, but one composed only of the preceding basis activities, to which proof of correctness techniques may readily be applied. Thus we see that the hardware representation maintains the linguistic constructs considered in Chapter II.

The second objective we had wished to attain was to reduce compiler complexity. Returning to our reasoning of Chapter I, if we consider the tradeoff between machine design and compiler writing in making a high level language operational, then we wish the compiler writer to do that which is easy to program and the machine designer to do that which is easy to build while maintaining the maximum semantic power. The desired result is to produce a simpler compiler which can achieve the same goals (as an equivalent compiler on a conventional machine) or allow the compiler
to model new semantic constructs without becoming correspondingly more complex. This particular savings is illustrated most vividly by considering the hardware representation of the program tree and its evaluation. On conventional computer architectures a compiler for a high level language may be viewed as performing two basic tasks. The first is the lexical analysis of the input text to form a program tree; the second is the process of code generation or interpretation. If the machine is one whose primitive instructions support the constructs of the language then code generation suffices; if more sophisticated semantic constructs requiring software support are used then generally an interpreter is required. By providing a hardware representation which evaluated a program tree a compiler may be produced whose sole task is to recognize and construct the tree for a particular program. This has two immediate effects. The first is that the elimination of the code generator allows the reduction in the size and complexity of the compiler to approximately half of that required on machines with linear instruction sequencing. The second, less obviously beneficial effect, is the requirement that those functions normally performed during code generation or interpretation be done either at
compilation or execution. The following paragraphs illustrate how the linguistic constructs introduced in Chapter II provide an effective and efficient division of labor.

The basic function of the code generator is to produce linear code from the program tree by making semantic interpretations upon the constructs being transformed. If the code generation is to be bypassed then the corresponding semantic information must be inherent in either the program tree or the evaluation mechanism, hence the semantic interpretations required must be made either when the program tree is constructed or at the time of evaluation.

The first question of interest is that of symbol table manipulation in the compiler and the semantic treatment of variables. In conventional machines without the use of values of mode name the compiler is required not only to build and maintain tables of the symbolic references used within each environment, but during code generation to model the environment entry and exit and to convert symbolic references to locations within the framework of that model. The use of runtime environment manipulation (allocate and call operators) and the use of elements of mode name for
symbolic reference from within the program permits the construction of compiler symbol tables with no address calculation. Thus symbolic references may be maintained throughout, even to the generation of hardware instructions. Note that only names are known at compile time. A location is not generated until execution of the appropriate take operator. Thus only reference variables may have values of mode \texttt{loc}, symbolic references from within the static program algorithm must be elements of mode \texttt{name}.

We next consider mode coercion. The first coercion is dereferencing. This coercion is provided directly at a hardware level by repeatedly applying the valuate operator to the operands of a primitive operator immediately prior to the application of that operator. Thus it becomes unnecessary for the programmer to explicitly specify indirection when manipulating primitive values through the use of reference variables. The explicit operators locate and valuate are provided for cases where it is necessary to manipulate reference variables themselves or achieve variations from the normal binding.

A limitation of the current implementation is illustrated by deproceduring. In the present model a call operator is required, hence deproceduring must be done
explicitly by the programmer or as a compile time activity. Noting the analogy with dereferencing we see that the results achieved in that area may be used to formulate a model for deproceduring. In the first place we see that procedure closure occurs when a procedure valued variable is stacked for evaluation. By maintaining as a literal subfield the number of environment accessed to reach the proper parent environment, closure may be delayed until application of the call operator. To permit earlier binding a close operator may be applied to convert the literal into a reference to the proper actual environment. Further, application of a primitive operator to an operand of mode \texttt{proc} may be programmed to generate an automatic procedure call, similar to the generation of valuative for references. While this would require greater information for the resumption of the formula from which the call is made, it seems a reasonable extension to the present machine.

The final coercion we consider is that of mode (data type) agreement for primitive operators and assignment. In high level languages on conventional untagged machines data type agreement is solely a compile time task unless a software interpreter is used. The present machine may be
examined at two levels: that at which it treats simple values, and that at which it treats aggregations and unions of values. Multiple mode arithmetic, mode agreement for precedence and assignment, modenames, and use of the conformity operator provide complete hardware representation, recognition, and manipulation of simple values. By the use of an indirect mode tag on row, name, and loc values mode checking is facilitated as a compiler activity for multiple values and unions of values. Extensibility is provided by user mode and the convert operator. The preceding paragraphs indicate how mode representations provide a powerful mechanism for the manipulation of values. Thus we see how the linguistic constructs developed in Chapter II allow the simplification of compiler design while at the same time extending the semantic range of the language implemented.

The third objective we had set was to maintain or improve the space utilization and the execution speed of the running program. Since we may consider memory a resource with a uniformly decreasing cost and time a resource with uniformly increasing (or at least uniformly nondecreasing) cost, we tend to bias speed of execution or semantic power against memory usage. The most obvious example of
this tradeoff is the model of the locus of control at the hardware level. This requires the use of a stack length is equal to the maximum statement nesting depth of the program. In a highly recursive program this could quickly become the major resource requirement. However, we note that use of the control locus allows a simple hardware environment manipulation strategy and multiple statement exit, and provides a complete history of the current control path for diagnostic purposes. Hence the more powerful semantic usage is well worth the space required. We no wish to show that program evaluation occupies a memory space and occurs within a time frame comparable to that of conventional machines.

We first consider iteration and selection statements. Figure V-1.a and b give the representation for a while statement in the high level language machine and a typical conventional machine respectively. In addition to the code required for the body and the predicate of the statement each machine requires two additional instructions. The while instruction reference itself is not counted here as it is the argument of a containing statement and it is counted in that statement. Figure V-1.c and d give the control flow for each machine. The major difference is a
stacking operation applied twice in the high level language machine prior to an instruction fetch. If we make the assumption that the algorithm region and execution record are on separate busses then the operations can be done in parallel and the times are identical. If stacking were done to high speed cache memory, speed could be improved to better than the conventional machine. However, even with a single bus structure the total additional cost of the high level language machine is two stack references per iteration cycle.

Consider next the repeat statement illustrated in Figure V-2. This is a more expensive construct to model as the ordering of the body and the predicate of the statement is ideal for linear instruction sequencing. The effect this has is to make while and repeat equally expensive constructs whereas repeat is easier by half on conventional machines. Again, what is gained in return for the additional burden is the semantic capability of maintaining a complete control locus at any point of program evaluation.

Next is the if statement illustrated in Figure V-3. We note that the high level language machine requires three instructions compared to two for a conventional machine. The flow graphs show that as in the case of the while
statement, the execution times are almost identical. Similarly, we consider the case statement illustrated in Figure V-4. In this particular instance the high level language machine requires only half as many instructions for implementation of the construct, but again execution times are very nearly the same. The final statement is the sequence, which has no direct analog in conventional machines which are intrinsically linear. Thus we see that the cost is \( N \) instructions for a statement of \( N \) clauses, and a time cost of four stack references per clause in the sequence.

We finally consider formula evaluation. We first consider the effects of the various operators. In the case of the call operator the copying of all parameters and environment manipulation is within the evaluation mechanism, hence represents a clear savings over any coded instructions or software service required for block or procedure entry on a conventional machine. The allocate operator has no analog on a conventional machine, however an equivalent amount of code must be generated to accomplish allocation at block entry on the conventional machine. Further, the allocate operator allows a more dynamic model of variable binding. ROWing operators require the same
resources as a conventional machine provided with segment descriptor addressing. Calculation and assignment of simple values require the same resources as a conventional machine. Only runtime mode checking for the assignment of structures would require software service.

To illustrate formula evaluation an example is taken from the program of Figure III-1. The following formula is considered.

\[
\text{filefree[file]} = \text{diagfree[rank-file]} = \\
\text{skewfree[file+(rank:=rank+1)]} = \text{true};
\]

Evaluating the source of the assignments first, we obtain the code in Figure V-5 a and b. The comparison is with a typical computer architecture containing an accumulator, a base register, an index register, and a operand stack. In both cases about twenty instructions are required. However, we note that in the case of the high level language machine a formula element is either a operator or an operand, while in the case of the conventional architecture it is an instruction which contains both an operation and an address field. Thus the high level language machine representation is more compact. To test this conclusion the program was written in PL/1 for an IBM 370/155. The same construct shown above when compiled using the PL/1
optimizing compiler required 216 bytes or 54 words of instructions.

Thus it has been shown in the preceding paragraphs that the operation of the high level language machine is comparable to that of a conventional machine, and in some areas far exceeds it. But the conventional machine used for comparison has been a relatively bare one with little semantic protection, while the very design of the high level language machine makes semantic verification mandatory. Thus the comparison has not been between two equivalent machines, but is biased strongly in favor of the conventional machine. If subscript range checking, mode checking, etc. are provided on a conventional machine then additional code must be generated, meaning that an equivalent program would be an order of magnitude larger and slower that on the high level language machine. At the same time the compiler writer is presented with a more formidable task in generating the semantic checks necessary. Hence the high level language machine diminishes the compiler writer's task while allowing more powerful semantics in a machine comparable in performance with conventional design. Thus we have accomplished the third objective.

The final goal considered in Chapter I was to reduce
interactive complexity. As noted at that time the problem was that the code generation process tended to obscure the semantic intent of the original program, so that the generation and interpretation diagnostics is a more difficult task. In contrast, consider the error conditions recognized by the control algorithms of Chapter IV. All invalid constructs are stated in terms of the linguistic constructs developed in Chapter II. At the time of the error a complete snapshot of the execution record is available, including the offending value, the processor state, universe of discourse of the program, and the complete control path from the root of the program tree to the current control point with the values of any partially evaluate formulae contained therein. Since this information is available at any time during program evaluation, by allowing "breakpoint snapshots" a more powerful diagnostic technique than tracing could be provided for monitoring program behavior.

We have considered the goals set in Chapter I and shown that the computer architecture developed as a refinement of the linguistic constructs of Chapter II provides a powerful semantic model while reducing compiler complexity by one half, maintaining performance range, and providing more transparent diagnostics. We now wish to consider the
semantics of high level language programming constructs and their relation to the computer architecture developed.

The relation between proof of correctness techniques and program structure was dealt with in Chapter II. The implementation of statements preserves this structure. In this chapter we have considered a number of areas. Hardware mode recognition and manipulation provides a direct representation of simple coercions and allows coercion of aggregations and unions of values as compile time or interpretive activities. Hardware dereferencing (and in the extension, deproceduring) allows a maximum of binding freedom within a stack evaluation. Finally, the use of mode-names and the allocate and conformity operators permit the manipulation of value classes as well as the actual values themselves. Thus we see that as well as providing a simple execution for current languages the semantic constructs of advanced languages are modeled. The following paragraphs consider the limitations in the implementation of the constructs modeled by the machine, suggest extensions to the model and areas of further research, as well as provide a rationale for the particular constructs chosen.

We first consider the area of statement evaluation.
Having first chosen to follow the precepts of structured programming and use selection, iteration and sequencing as the basis activities for statement construction, we are faced with the question of what actual statements to provide. Those statements outlined in Chapter II semantically over the basis activities and provide an efficient hardware implementation of common multiple constructs. However, we note that the form:

    for "variable" from "formula1" by "formula2" to "formula3" do "clause" od;

although a common programming language construct, does not have a direct hardware representation. However, we note that the form:

    take int l:="formula1"-"formula2";

    while if "formula2" 0 then
        (i:=i+"formula2") "formula3"
    else
        (i:=i+"formula2") "formula3"
    fi
    do "clause" od;
    free i;

is semantically equivalent and has a direct representation. The if is required since the incrementation may be either
positive or negative. If the sign of the increment can be determined at compile time if the if may be removed. Thus we see that in the context of the machine architecture the for and while constructs are sufficiently redundant for one statement to implement both constructs.

A second construct of interest is the case statement. In some high level languages (notably BCPL, in which the simulator was written) the case statement is implemented as a jump into a sequenced block rather than a strictly indexed choice. Thus once within the case statement all remaining subclauses after the chosen one are executed unless the control flow is subsequently altered. This construct is one whose addition would cause little additional complexity to the machine. By converting a case to a sequence at the time the predicate is tested and restacking the reference, the task is accomplished. This would provide a simple extension to the model and since there is no clear indication which is the more useful construct, implementing both would provide data about usage and preference.

The next area we consider is formula evaluation. The basic implementation restriction is stack evaluation of temporaries within a formula. This provides bound
restriction on operations on row values (no expansions) and on the ordering of operand evaluation. Since compile time ordering of operands is required postfix notation is used since it delays operator application. Thus by allowing binding to occur at operator application and providing explicit binding operators considerable semantic power can be achieved in the translation of references to their respective values. Thus we seem to have achieved the maximum semantic power available within a stack evaluation framework.

The next question we consider is that of dynamic variables. The semantic model provides for an associative memory for named variables and a heap region for multiple values. Thus we see that storage exchange, hence software service is required for allocation, freeing, and garbage collection or reference count management. However, software support is not required for the binding of references as it is for dynamic variables on conventional architectures.

Finally, we consider a possible extension to the model concerning the notion of functional variables. The notion is: given a variable x, provide as a named variable some function of x (say: 2*x+1) in any environment in
which \( x \) is defined. A natural extension to the high level language machine encompasses this construct. Since statements have been defined to return values, they may be when evaluated as objects which possess a value. Thus it makes sense to include them in the universe of discourse. This is accomplished by allowing variables of mode clause, whose values are statements or formulae. Thus evaluation becomes another binding operation within the environment in which the clause variable is contained. Further work is required to determine side effects and implementation restrictions.

This thesis has been concerned with the design of a language directed computer architecture. A simple simulation has proven its feasibility vis a vis conventional machines. Viewing the long range goal of experimentation with a working implementation (simulation or machine) a number of projects present themselves. The most obvious is a compiler. Since the architecture has been designed at a linguistic level, a compiler is the most suitable test of the machines capabilities. Implementation of an Algol68 or LISP like language would prove instructive. This provides the second project. For dynamic allocation storage management is required. A storage exchange package to administer the heap area would be a needed task. Finally
the architecture has been designed solely as a language processor. Operating system research is required including possible hardware extensions to the machine itself.

In summary we see that a process of stepwise refinement analogous to Dijkstra's program refinement has been used to design a high level computer architecture; starting with a set of linguistic constructs, leading through a dynamic process model to a machine architecture at a register transfer level. Hence the refinement process maintains the structure of the programming linguistics. Thus the machine produced, while roughly comparable space-while and timewise to conventional architectures, provides a more powerful and coherent semantics, thus allowing more expressive languages to be implemented more easily and efficiently.
Figure V-1

***While:***

```
X:  X:
```

Predicate Body

a

---

**Loop:**

- Predicate
- Jump(false) End Body
- Jump Loop
- End: ...

b

---

**Enter**

- Stack instruction
- Fetch instruction
- Evaluate Predicate
- Unstack instruction

true

- Stack instruction
- Fetch instruction
- Evaluate Body
- Unstack instruction

false

Enter

---

c

---

**Enter**

- Fetch Instruction
- Evaluate Predicate
- Fetch instruction

true

- Fetch instruction
- Jump
- Exit

false

---

d

---
Figure V-2

\[ \begin{align*}
\text{Repeat:} & \quad \ldots \\
\text{Loop:} & \quad \text{Body} \\
& \quad \text{Predicate} \\
& \quad \text{Jump(false) Loop} \\
& \quad \ldots \\
\text{Body} & \quad \text{Predicate} \\
\end{align*} \]

\[ \begin{align*}
\text{Enter} & \quad \text{Stack instruction} \\
& \quad \text{Fetch instruction} \\
& \quad \text{Evaluate Body} \\
& \quad \text{Unstack instruction} \\
& \quad \text{Fetch instruction} \\
& \quad \text{Evaluate Predicate} \\
& \quad \text{Unstack instruction} \\
\text{Exit} & \quad \text{true} \\
& \quad \text{false} \\
\end{align*} \]

\[ \begin{align*}
\text{Enter} & \quad \text{Fetch instruction} \\
& \quad \text{Evaluate Body} \\
& \quad \text{Fetch instruction} \\
& \quad \text{Evaluate Predicate} \\
& \quad \text{true} \\
& \quad \text{false} \\
\text{Exit} & \quad \text{Jump} \\
\end{align*} \]
Figure V-3

(a) If:

Predicate
Then
Else

(b) Jump(false)
No
Then
Jump
End
Else
No:
End:
...

(c) Enter

Fetch instruction
Evaluate Predicate
Fetch instruction
true/false

Enter
Stack instruction
Fetch instruction
Evaluate Predicate
Unstack instruction
Then
Jump
Else
true/false

Exit

(d) Exit

Fetch instruction
Evaluate Then
Jump
Exit
Evaluate Else
Figure V-4

Case:

Predicate Choice0...Choice#

Enter

Stack number
Stack instruction
Fetch instruction
Evaluate Predicate
Unstack instruction
Unstack number
Address calculation
Fetch instruction
Evaluate Choice_i

Exit

Predicate
Compare TEMP,#
Jump(leg) Go
Clear TEMP
Go:
LoadI TEMP
LoadB JTABLE
Jump (B+I)
JTABLE:
L0
...
L#
L0:
Choice0
Jump End
...
L#:
Choice#
End:
...

Enter

Fetch instruction
Evaluate Predicate
Address calculation
Fetch label
Jump
Fetch instruction
Evaluate Choice_i
Jump (unless i=#)

Exit
Figure V-5

21, bool: true, name: rank, int: l, -, name: rank, :=, name: file, +, name skewfree, index, :=, name: rank, name: file, -, name: diagfree, index, :=, name file, name filefree, index, :=

a

Stack true
LoadB @rank
LoadA (B)
Dec
Sto (B)
Add file
LoadB @skewfree
LoadI A
Unstack A
Sto (B+I)
Stack A
LoadA rank
Sub file
LoadB @diagfree
LoadI A
Unstack A
Sto (B+I)
LoadI file
LoadB @filefree
Sto (B+I)

b
BIBLIOGRAPHY


10. Rice Computer-2 General Specifications, Rice University, Houston.

11. IBM System/370 Principles of Operation, IBM Corp., Poughkeepsie, N.Y.


30. The Treatment of Data Types in EL/1, Comm ACM 17,5(May 74), 251-264.


34. Notes from a course on MADCAP VI at Los Alamos Scientific Laboratory, 23-31 August 1972.


Additional Readings


Church, A., The calculi of lambda-conversion, Annals Math Studies, no. 6, Princeton University Press, 1941.
