THE REPRESENTATION AND TRANSFORMATION OF FUNCTIONS

Peter Wegner

Technical Report

No. 68-13

April 1968

Department of Computer Science
Cornell University
Ithaca, New York 14850

THE REPRESENTATION AND TRANSFORMATION OF FUNCTIONS

Table of Contents

	Page
Abstract	1
1. Basic Concepts	2
1.1 Function Representation	2
1.2 Information Structure Models	5
1.3 Examples of Information Structure Models	11
1.4 Environments, Parameters and Instantaneous Descriptions	20
1.5 Declarations and Binding Time	25
2. Functions Represented by Symbol Tables	36
2.1 Encoders and Assemblers	37
2.2 Macro Assemblers	41
2.3 Generalized Macro Systems	44
2.4 Attributes of Symbol Table Entries	48
3. The Lambda Calculus	54
3.1 Syntax	58
3.2 Semantics	59
3.3 Values and Value Classes	62
3.4 Lambda Calculus Machines	69
4. Information Structure Models for Procedure Oriented Languages	83
4.1 Function Modules in ALGOL	83
4.2 Environment Management in ALGOL	87
4.3 Models for Programming Languages	96
Conclusions	97
References	99

THE REPRESENTATION AND TRANSFORMATION OF FUNCTIONS*

Peter Wegner

Cornell University

April 1968

Abstract

A new approach to the study of programming languages is developed, which emphasized the sequence of information structures generated by a program during its execution. In section 1 a class of models called information structure models is developed for characterizing computations as sequences of transformations of information structures. The notion of binding time is introduced and different binding strategies for programming languages are considered. Section 2 considers the representation of functions by symbol tables and analyzes a number of examples of such functions, such as assemblers and macro systems. Section 3 introduces a very simple programming language for function evaluation known as the lambda calculus, and considers information structure models for alternative evaluation strategies for the lambda calculus. Section 4 shown that ALGOL computations can be specified as information structure models with the same basic characteristics as lambda calculus computations but with a richer set of primitives, and indicates how languages such as PL/I can be characterized as information structure models.

---

*First draft November 1966, revised August 1967, March 1968.

THE REPRESENTATION AND TRANSFORMATION OF FUNCTIONS

Peter Wegner

Cornell University

April 1968

Introduction

This paper develops an approach to the study of programming languages starting from the notion that a program with its data constitutes an information structure and from the notion that a computation is represented by a sequence of information structures generated from an initial representation by the execution of a sequence of instructions. In section 1 a class of models called information structure models is developed for characterizing computations as sequences of transformations of information structures. Section 2 considers the representation of functions by symbol tables and analyzes a number of examples of such functions. Section 3 introduces a very simple programming language for function evaluation known as the lambda calculus, and considers information structure models for alternative evaluation strategies in the lambda calculus. Section 4 shows that ALGOL computations can be specified as information structure transformations with the same basic characteristics as lambda calculus computations but with a richer set of primitives, and indicates how languages such as PL/I can be characterized as information structure models.

The approach to programming language specification developed in the present paper yields insights into the nature of programming language semantics and provides tools for the analysis and characterization of programming languages. Some of the models described here are developed in greater detail in reference 1. The paper is written in an expository style, but contains a number of original concepts.

1. Basic Concepts

1.1. Function Representation

A program in a programming language may be thought of as a realization of a function f which, given a *data set* x to which it is applicable, specifies a *value* $y = f(x)$. The value y is said to be *computed* by *applying* the function f to its data.

Functions are usually defined by mathematicians as rules of correspondence between arguments and their values.

Definition 1: A function f is a rule of correspondence between a set X of arguments (data elements) and a set Y of values (results), which for every argument $x \in X$ specifies a value $y \in Y$. The set X is said to be the *domain* of the function. The set Y is said to be the *range* of the function. The value y associated with a given x is denoted by $y = f(x)$ and is said to be obtained by *application* of the function f to the argument x.

Nothing is said in the above definition about the representation of the function f. Since the function is defined as a rule of correspondence, the most obvious representation of a function is by means of a set of ordered pairs (x,y) containing precisely one pair for every $x \in X$.

Since the number of elements in the domain X may in general be very large or even infinite, the uniform representation of all functions by lists of ordered pairs becomes impracticable. Functions are instead specified by programs which prescribe how the value of y may be computed for any

element x in the domain of the function.

When considering the mechanical evaluation of functions on a computer, both the program and its data must be represented in the information storage medium of the computer in a form that is amenable to computation. Evaluation consists of applying the representation of the function f to the representation of the data x in order to compute a representation of the value y .

A function is usually thought of as a more active component in a computation than the data on which it operates. However, during a computation, both the function representation and the data representation are treated as passive components. The active component responsible for applying a function to its data is called a processing unit or a processor. During a computation the processor scans a sequence of components of the function representation (instructions) and performs a sequence of transformations of the function and data representation as illustrated in figure 1.

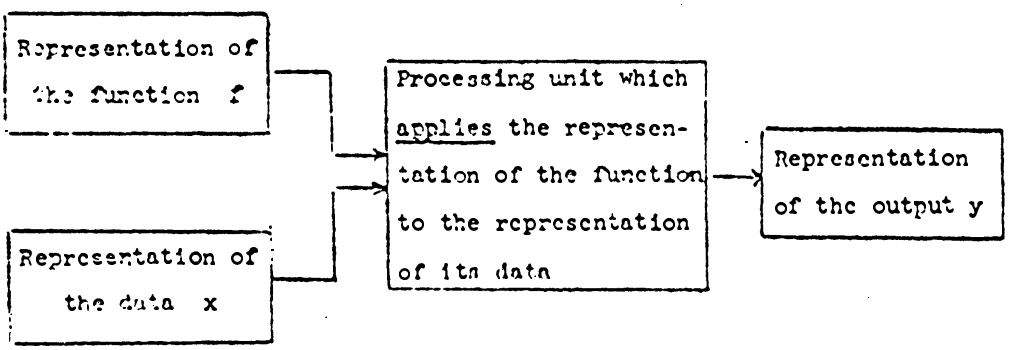

Figure 1. Application of a Function to its Data.

In figure 1, the representation of the function f is separated from the representation of its data x , and both are separated from the representation of the value y . However, in the general case, there is no rigid distinction between functions, data and output, and it is more appropriate to think of the representation of a function together with its data as a single structure which is progressively transformed into a representation of the output.

The current literature on programming languages tends to emphasize

a) The static structure of programs in a programming language and

b) The translation of programs from a source language to a target language.

In the above terminology area a) is concerned with alternative initial representations of programs, and area b) is concerned with transformation from one initial representation to another. Neither area a) nor area b) deals with the process of function evaluation itself, but merely with initial representation of information prior to a computation.

The study of transformations that occur during function evaluation should play a more central role in programming language theory than the study of translation from one initial representation to another. It will be shown that the overall characteristics of such transformations are relatively insensitive to changes of initial representation. This means that the characteristics of the evaluation process for a wide class of programming languages can be described in a relatively machine-independent manner.

An understanding of how programs with their data are represented and transformed during their execution yields a direct understanding of how computers which execute such programs operate. This approach to programming languages therefore has greater relevance to machine organization than the study of translation processes or of static program structures.

1.2 Information Structure Models.

Programs together with their data are examples of information structures. An information structure is defined as a finite string of characters over a finite alphabet T. We shall be concerned not only with individual strings over an alphabet T but also with sets of strings over an alphabet T. The set of all strings (information structures) over an alphabet T will be denoted by T^*. The sets of strings considered are assumed to be effective (recursive)* subsets of the set T^* of all strings over the alphabet T. An effective set $L \subseteq T^*$ is sometimes called a language. Programming languages are examples of effective sets $L \subseteq T^*$, where T^* is the alphabet of the programming language. Sets L whose elements are programs with their data also constitute effective sets L over an alphabet T^*.

A computation can be specified independently of a digital computer as a sequence of transformations of an initially specified information structure. The following terminology is useful for talking about information structures at different points during the execution of a computation.

---

*A subset $L \subseteq T^*$ is effective (recursive) if and only if for any element $s \in T^*$ it is possible to determine whether $s \in L$.

Definition 2. A snapshot of the complete information structure associated with a computation at a given instant of time will be referred to as an instantaneous description. It is assumed that instantaneous descriptions can be associated only with discrete time points corresponding to the beginning or end of a primitive instruction.

Definition 3. A computation consists of a sequence of instantaneous descriptions, where one instantaneous description is obtained from the previous one by execution of an instruction. The first instantaneous description is referred to as an initial representation. The last instantaneous description is said to be a final representation. Representations which are neither initial nor final are said to be intermediate representations.

One of the principal objectives of the study of programming is to characterize the structure of instantaneous descriptions for classes of programming languages, and to characterize the way that instantaneous descriptions change as programs are executed.

A programming language characterizes a class of computations corresponding to the set L of all programs of the programming language together with their data. A model which defines a class of computations such as those determined by a programming language will be called an information structure model.

Definition 4. An information structure model is a pair (I,F) where I is a set of information structures referred to as initial representations; and F is a set of transformations of information structures referred to as primitive instructions.

A computation in an information structure model is a sequence $I_o \xrightarrow{f} I_1 \xrightarrow{f} I_2 \cdots \xrightarrow{f} I_n$, where $I_o \in I$ is an initial representation, I_n is referred to as a final representation, and each $I_j \rightarrow I_{j+1}$, $j = 0,1,2,\ldots,n-1$ is accomplished by an element $f \in F$. The final representation I_n has the property that no elements of F are applicable to it.

Definition 5. For a given information structure model (I,F) the closure $cl(I)$ of I is defined as the set of all information structures which can be generated from I by finite sequences of operations of F.

Operations $f \in F$ may be thought of as unary operations over $cl(I)$. However, an operation f rarely uses more than a small subset of the information fields of the instantaneous description I_j that constitutes its argument, and rarely modifies more than a small subset of the information fields of the instantaneous description I_{j+1} that constitutes its value. The set of information fields required by f to determine its action will be referred to as the domain of f. The set of information fields modified by f will be referred to as the range of f.

An instantaneous description U_j for a given class of computations

(I,F) can be characterized by specifying its components. For example the components of an instantaneous description I_j for a given programming language might include the program component, input and output components, temporary storage structures such as stacks and symbol tables, and a stateword component which contains a pointer to the current point of execution. The operations $f \in F$ may be described in terms of how they transform each of the components of the instantaneous description.

Each component of an information structure may in turn be described in terms of lower level components until a set of primitive information components is reached. Detailed specification of an information structure model (I,F) requires a syntactic notation for specifying the set I and cl(I), and a semantic notation for specifying the transformations determined by F. Currently used syntactic notations include the notation of formal grammars (2)(3), and the notation for specifying analytic syntax developed by McCarthy.(4) Currently used semantic notations include those developed in connection with compiler compilers (5)(6)(7)(8), those developed for the theory of computation,(4) and those developed for automata theory.

A language for specifying information structure models will be called a modelling language. A modelling language must have facilities for specifying the syntax of instantaneous descriptions, and facilities for specifying transformations when structures with a given syntax are encountered. An implementation of a modelling language is similar in concept to a compiler compiler. In both cases a specification of syntax and semantics

of a specific language permits handling of programs in the specified language. During execution both systems perform a sequence of actions triggered by recognition of syntactic structures. However, the actions of a compiler compiler are directed to compilation while the actions of an information structure modelling system are interpretive transformations. An information structure model (I,F) specified in a modelling language defines a syntax directed interpreter for initial representations I. The programming system which implements the modelling language may be called an interpreter interpreter, since it is an interpreter that executes programs which are interpreters. Work on the development of an interpreter interpreter is currently in progress.

The complete characterization of an information structure model requires considerable technical detail and is unduly machine dependent. We shall emphasize qualitative descriptions of information structure models for a number of classes of function representations in terms of their higher level information components. One of the advantages of the information structure model approach is that considerable insights and understanding of classes of computations can be obtained by modelling higher level components without specifying the detailed representations and transformations at lower levels.

The higher level characteristics of an information structure model are relatively insensitive to compilation. Most compilation processes preserve the identity of operators and operands and the order in which operators are

are applied to operands. For such compilations there is a correspondence between information fields of the source and target representations, and a correspondence between information fields of intermediate representations for corresponding computations. If C' is the information structure model for the compiled version of C , then a correspondence may be established between initial information structure components of C and C' , and the order of creation, deletion and modification of information fields corresponds for the two models. A description of function evaluation which emphasizes creation, deletion and modification of information fields is insensitive to compilations satisfying the above conditions.

Although the information structure model approach deemphasizes the role of compilation, the characteristics of a specific compiler or a class of compilers may fruitfully be studied by specifying an information structure model for compilation. The execution time characteristics of compilation processes form an interesting subject of study. However, a compiler for a given language does not tell us anything essential about the class of computations whose representation it translates but merely about the correspondence between two representations of that class of computation. Much of the confusion about the semantics of programming languages that occurs in the literature is due to the fact that the semantics of compilation has been confused with the semantics of execution.

1.3. Examples of Information Strueture Models.

Information structure models can be used to describe classes of computation defined by programming languages, computers and Automata. Information structure models are developed below for the following systems.

a) Finite automata, Linear Bounded automata and Turing machines.

b) Pushdown automata, stack automata and Balloon automata.

c) Digital computers.

d) Programming Languages.

The models a) through d) are progressively more complex. Simple models will be developed in detail while more complex models will be developed in terms of the transformation characteristics of higher level information components. Information structure models are concerned with describing classes of computations rather than with proving theorems about them. The models for automata below do not materially contribute to the development of automata theory, but provide the programmer with insights into the relation between classes of computations defined by models of automata, computers and programming languages.

a). Finite Automata, Linear Bounded Automata and Turing Machines.

A finite automaton is characterized by a finite set S of states and a finite input alphabet T. During any computation it starts from an initial state s_0 and operates on an input tape which contains a finite string of characters of T, say $t_1 t_2 \ldots t_n$. We shall assume that tape squares are numbered $1,2,\ldots,n$ and that tapes are bounded by the symbol

which is disjoint from the aphabet T , as indicated in figure 2.

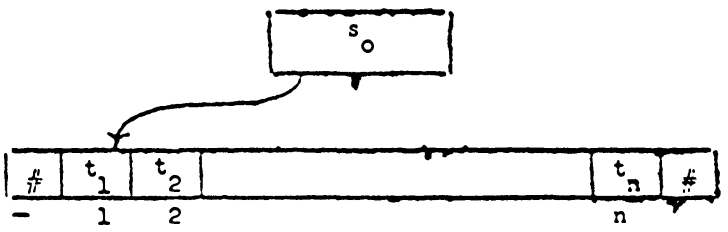

Figure 2. Information Structure Associated with a finite Automaton at the Beginning of its Computation.

The set I of initial representations associated with a finite automaton can be characterized by triples of the form $(s_o) \times T^* \times (1)$. i.e. Triples which specify the initial state s_o , a tape $\tau \in T^*$, and the integer 1 , specifying that we are pointing to the first square of the tape.

In order to complete the specification of the information structure we must specify the closure cl(I) of structures which may be generated during execution, and the set F of primitive operations that may act on the information structures.

The closure cl(I) is $S \times T^* \times N$ where N is the set of positive integers. The operations $f \in F$ all have the form $(s,t,k) \rightarrow (s',k+1)$, where $t \in T$ and $1 \leq k \leq n$. i.e. Scanning of the symbol t in position n and state s yields a new state s' and incrementation of k to k+1 . The transition from k to k+1 may be assumed to be an implicit rather than explicit part of f , just as incrementation of the

instruction location register is assumed to be an implicit rather than explicit part of the action of a primitive instruction on a digital computer. The elements $f \in F$ may be specified as $(s,t) \rightarrow s'$.

The transformation of the information structure that can be accomplished by a primitive instruction is severely limited. Each of the three components S , T^* and N of $cl(I)$ is subject to a different mode of access and transformation. Thus T^* cannot be transformed and may be accessed only in a read-only mode, N can be transformed only by incrementation, and there are no general restrictions on transforming elements of S into other elements of S .

The restrictions with regard to transformation impose a topology or graph structure on the subset of information structures which constitute instances of the component, such that there is a directed arrow from element x to element y of the subset if and only if there is an $f \in F$ which transforms x to y .

The graph of a read-only structure such as T^* for finite automata consists of a set of isolated points. The graph of a structure with no restrictions, such as the set S of states of a finite automaton, is a complete graph with an arrow from every element to every other element. The graph of the set N of tape positions has a directed arrow from each integer to its successor.

The above graphs are for the class of finite automata . For individual finite automata the graphs for T^* and W will remain the same but the graph for S will no longer be a complete graph.

Linear bounded automata and Turing machines have the same class of initial representations as finite automata but more powerful transformation rules F. A linear bounded automaton permits the information component T^* to be modified by writing on the square currently pointed to by the instruction pointer, and permits motion of the instruction pointer one position to the right or to the left. The instructions f are of the form $(s,t) \rightarrow (s',t',d)$ where t' is the symbol written in place of t, and d is -1, 0, or 1 and determines the tape motion. The linear bounded automaton stops if it moves onto an end of tape marker $\#$. i.e. There are no instructions in which the second component of the domain is $\#$.

Turing machines allow writing on the tape and extending the tape in either direction. Formally, we can think of all three models as having an initial tape consisting of a finite symbol string of T embedded in a two way infinite string of symbols $\#$. The difference between linear bounded automata and Turing machines can then be characterized by allowing Turing machines to have instructions of the form $(s,\#) \rightarrow (s',t',d)$. This class of instructions has the additional effect of enlarging the closure $cl(I)$ for Turing machines to be of the form $S \times T^* \times Z$ where Z is the set of integers.

b). Pushdown Automata, Stack Automata and Balloon Automata.

Finite automata, linear bounded automata and Turing machines have only a single information component of unbounded size. Pushdown automata, stack automata and balloon automata may have more than one unbounded information component. All three models may access the data tape component only in a read-only mode, and differ in having different restrictions on accessing the second information component.

Pushdown automata have an information component called a stack. Elements of the stack can be created and deleted only in a last-in-first-out order, and only the most recently created (top) element of the stack may be accessed. Stack automata are more powerful than pushdown automata in that they can access information in the interior of the stack in a read-only mode. Balloon automata have a second unbounded information component called a balloon. The specification of different accessing restrictions on the Balloon leads to different classes of automata.

Automata can be classified in terms of the richness of the classes of functions they can compute, and it can be shown that the class of functions which can be computed by Turing machines includes the class of functions computable by any Automaton or digital computer. In developing information structure models for automata and computers we are concerned not with classes of functions computable in the class of computations, but merely with describing the information structure transformations that occur when computations of the given class are performed.

c). Computers.

Consider first a computer having a processing unit PU and a memory M with a finite number of registers numbered 1,2,...,n each of which may contain an element t of a finite set T .

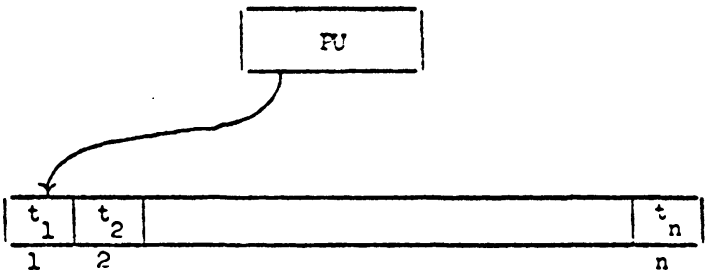

Figure 3. A Simple Computer.

If M has n elements (registers) then the set I of initial representations can be characterized as $PU \times [T^*]_n \times \{1\}$ where $[T^*]_n$ specifies the subset of strings of T^* with not more than n characters. cl(I) becomes $PU \times [T^*]_n \times \bar{n}$, where $\bar{n}$ represents the set of integers 1 through n .

Although the set I for the above model is restricted to be finite, the set of operations $f \in F$ is much richer than that for Turing machines. A primitive instruction t_i in a computer normally has an operand field which specifies a data element t_j in address j . An instruction domain thus includes not only the symbol (memory register) t_i being accessed but also a symbol t_j pointed to by t_i . If indirect addressing chains are permitted during execution then an arbitrary number of elements t may be accessed during the recognition phase. The transformation phase may likewise

be more complex than that for automata. Whereas Turing machines may transform only the field actually being scanned, computers may transform (store information into) any field of the memory. The instruction pointer may be incremented arbitrarily in computers and only by plus or minus one in Turing machines.

The greater versatility of primitive instructions in computers does not result in greater richness of functions that can be computed, since it has been shown that all computable functions are computable by Turing machines. Indeed there are functions computable by Turing machines which are not computable by any real computer because of its finite size. However, many functions which <u>are</u> computable on a given computer can be computed in a smaller number of steps than on a Turing machine. The greater richness of primitive transformations on an actual computer is designed to make computations more efficient.

Computers usually have a number of information components in addition to the memory and processing unit. These information components include one or more input components, one or more output components, and a hierarchy of auxiliary memory components. Each of the information components has a characteristic structure and mode of access determined by the primitive computer instructions that are applicable to it.

A digital computer is a "housing" for a class of information structures The computer permits mechanical transformations on information structures stored within it to be performed at a very rapid rate. However, the

information structure model associated with a computer is independent of physical hardware. A mechanical prescription for performing the transformations specified by the computer on the class of information structures which may be stored in the computer will serve in place of the computer, although it will of course be far less efficient. The information structure models for programming languages disoussed below correspond to mechanical presoriptions for perfoming classes of computations.

d). Programming Languages.

Programming languages may be represented as information structure models independently of the computers on which they are executed. As a first approximation assume that the class I of initial representations of a program consists of three components.

1. A stateword component W which contains information normally contained in the processing unit of a computer, including an instruction pointer.
2. A program component P .
3. A data component D .

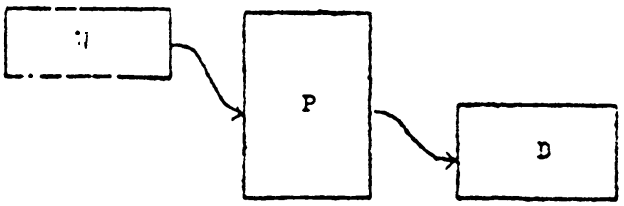

Figure 4. Information Components of a Program.

A primitive operation f can be specified in terms of the three information components (W,P,D) as $(W,P,D) \xrightarrow{f} (W',P',D')$. Each of the three components W, P, D has a characteristic structure and mode of access.

Transformations of the W component are usually restricted so that its size does not change during its execution. In early language implementations the P component contained modifiable instructions.* However, it has become accepted practice to program the P component to be reentrant, and to store instruction modifiers and other variable information associated with the program in the D component. When the program component is reentrant, then primitive transformations f are restricted to be of the form $(W,P,D) \rightarrow (W',P,D')$.

Programming languages may be characterized by the kinds of transformations permitted in the D component. FORTRAN is carefully designed so that the structure of the D component in cl(I) is fixed prior to execution. ALGOL is designed so that the D component is essentially a stack with last-in-first-out order of creation and deletion of information structures. List processing and string manipulation languages require

---

*When stored program computers were first developed, the possibility of modifying instructions as though they were data was hailed as a big breakthrough. It was [illegible] realized that modification of instructions leads to unmanageable program structure, and makes debugging and program modification more difficult. Program modification is simulated by hardware devices such as index registers.

the D component to take an even more flexible form, with unpredictable creation and deletion requirements. One method of organizing D so that information structures can be flexibly created and deleted, is to organize D as a (free storage) list structure from which blocks can be allocated and to which blocks can be returned.

1.4 Environments, Parameters and Partial Instantaneous Descriptions.

The active agent which performs transformations in an information structure model is called an interpreter. Interpretation always involves a recognition phase in which the information fields in the domain of the instruction are recognized followed by a transformation phase which transforms information fields in the range of the instruction. When the interpreter is a digital computer, information fields in the domain have a relatively fixed format. Machine language instructions usually have an operation field, one or more address fields and a number of auxiliary information fields. The domain of primitive operations F of a progamming language information structure model is usually more variable than that of a computer.

The set of all meaningful information structures that can occur in the instruction domain at a given point of the computation is referred to as the environment at that point of the computation. If S is the set of all elementary environment at a given point of the computation, and $t_i \in T$ is a specification of the transformation to be performed when an

element s_i of the environment is encountered, then the set of possible actions at the current point of the computation can be specified by a symbol table (s_i,t_i). Program execution may be thought of as being performed by successive symbol table look-up to find the element s_i corresponding to the current instruction followed by performance of the action t_i. In real computers the set of actions associated with primitive instructions is not explicitly stored in a symbol table, but is implicit in the logical design of the computer. However, in interpretive models of programming languages, the information structure transformations may in a very real sense be specified by such a symbol table.*

The symbol table in which correspondences between domains and actions are stored is usually thought of as an ordered symbol table. Instead of thinking of the symbol table as a list of ordered pairs we can think of it as a conditional expression of the form "if s_1 then t_1 else if s_2 then t_2 ... else if s_n then t_n". This is the form in which mechanical evaluation algorithms such as the LISP APPLY function (10) and the SECD machine (11) are specified. Syntax directed compilers such as that of Floyd (12) and Wirth and Weber (13) are also specified in this way. Both an interpreter interpreter and a compiler compiler consist

---

*The elements s_i of a programming language defined by a context-free grammar may be associated with reductions $X \leftarrow U$ where X is a nonterminal and U is a string over terminals and nonterminals. The elements t_i associated with a given s_i are translation-time actions in a syntax directed compiler and execution time actions in a syntax directed interpreter.

essentially of a mechanism for specifying ordered symbol tables of this kind. The principal difference between an interpreter interpreter and a compiler compiler is that the t_i components of an interpreter interpreter are concerned with execution-time transformations, while the t_i components of a compiler compiler are concerned with compile-time transformations.

Every instruction specification (s_i,t_i) may be thought of as a function whose domain and range are instantaneous descriptions of an information structure model. Let I be the set of all instantaneous descriptions in which (s_i,t_i) is a next instruction to be executed, and let I' be the set of all instantaneous descriptions that may result from the execution of (s_i,t_i). The function associated with (s_i,t_i) is a set of ordered pairs (I_B,I_A) where $I_B \in I$, and $I_A \in I'$ is the instantaneous description resulting from the application of the instruction (s_i,t_i) to I_B. The ordered pairs (I_B,I_A) are sometimes written in the form "$I_B \rightarrow I_A$" and referred to as productions. In actual information structure models each instruction has a restricted domain D and range R, and the effect of (s_i,t_i) is specified directly in terms of the information fields of the domain and range as $s_i \rightarrow t_i$.

It is convenient to allow domain elements s_i of an interpreter to specify groups of similar transformations rather than merely a single transformation. This can be accomplished by associating parameters with elements s_i and allowing the action t_i associated with s_i to depend on the values of parameters associated with s_i. When the interpreter is

a digital computer (say a one-address computer) and elements s_i are instructions, then the parameters of a given s_i include the address field, the content of the address field, and fields that are accessible to the instruction without being explicitly specified such as the accumulator. Parameters of an element s_i may in general include explicit parameters such as address and index register fields which are explicitly specified in the instruction format and implicit parameters such as the accumulator and instruction pointer which need not be explicitly specified. Parameters may themselves have implicit parameters associated with them, resulting in complex interpretive actions during parameter evaluation.

The notion of a parameterized domain s_i arises not only at the level of primitive instructions but also at many other levels in specification of an information structure model. For example a computer C can be thought of as an interpreter which requires two explicit parameters P (machine language program) and D (data in the domain of P) to completely specify its action $C(P,D)$. The stateword which specifies the first instruction to be executed is assumed to be an implicit parameter.

An important notion arising for parameterized interpreters is the notion of a derived interpreter that is obtained when some but not all of the parameter values are supplied. This notion is familiar for functions. For example the device of turning the two-parameter function + into the one parameter successor function (+1) by fixing one of the parameters is well known. A similar device can be used to turn a parameterized interpreter C into a different interpreter C' by fixing some of

the parameters of C.

If the interpreter C is a computer requiring the two parameters P and D, then fixing of the first argument P turns the interpreter C into an interpreter C_p which may be thought of as a device for mechanical interpretation of elements of the data domain D of the program P. When the programs P of a given programming language are equated with the functions they denote, this is justified precisely because of the existence of a computer C which, when loaded with any program P of the language, simulates the function C_p.

A representation of a function together with its data will be referred to as a complete instantaneous description, while a representation of a function without its data will be referred to as a partial instantaneous description. A partial instantaneous description is said to be completed by supplying a representation of the data component. There is a one-to-one correspondence between representations of data in the domain of the function and completions of the instantaneous description.

More generally a parameterized interpretation domain s_i is said to be partially specified if only a subset of the information structures required to perform the transformation t_i are specified. Every partially specified interpretation domain s_i defines a new parameterized interpretation function.

When P is a problem program, the domain of C_p is restricted to a limited class of information structures. However, the class of information

structures which constitutes the domain of C_P may be just as rich as that of C . In particular if P is an interpreter, then it converts C from a computer which accepts programs with their data in one programming language into a computer which accepts programs with their data in another programming language.

When the class of partial instantaneous descriptions of a given computer C permits the specification of any computable function, then C is said to be a universal computer. A universal computer U may be converted into another universal computer U' by means of a program P which is a partial instantaneous description of U and has the domain U' . The program P is an interpreter which allows programs written in the machine language of U' to be interpretively executed on U , and is said to simulate the computer U' on the computer U .

1.5 Declarations and Binding-Time

In procedure oriented languages there are usually facilities for creating information structures during execution and for assigning names to created information structures. Creation and naming of an information structure is performed dynamically during execution just like assignment of a value to an information field. However it is convenient to distinguish information structure creation from value assignment to fields of the information structure. Creation of an information structure will be referred to as a declarative action, while assignment of a value to an information field will be referred to as an imperative action.

A declarative action may be thought of as creating an entry (s_i, t_i) in the environment where s_i is the name of the entry and t_i is the action of accessing the information structure. If t_i has a complex internal structure, then s_i may be parameterized to allow selection of subfields of t_i and specification of other attributes of t_i.

It is sometimes convenient to further subdivide declarative actions into template specification actions and structure creation actions. A template specification action defines a class of information structures by means of a template, and may specify transformations for data structures of this class. The name associated with a class of information structures created by a template is said to be the type defined by the template.* Procedure oriented languages usually contain a number of primitive types which may be thought of as being specified by system templates. Languages which permit specification of new structures by means of templates and specification of the operations applicable to the class of structures

---

*The complete specification of a type requires both a template which specifies a class of information structures, and specification of the operations applicable to information structures of the given type. A type specification in an information structure model is effectively an information structure model for a set of substrings of the model in which it is embedded. Type templates are frequently chosen so that the information structures associated with each type are not disjoint. In this case the type of a given information structure is determined by the context in which it occurs during the computation.

defined by a template are said to contain definitional facilities.

A template specification action creates a symbol table entry (s_i,t_i) in the environment, where s_i is the name of the type and t_i is the action of creating an instance of the type s_i. A structure creation action for a given template (s_i,t_i) creates a further symbol table entry (s_j,t_j) where s_j is the name of the created instance of the template and t_j is the created information structure. Both template specification and structure creation modify the environment, but in different ways.

The distinction between declarative and imperative actions is essentially a device for breaking down the specification of an information structure into multiple stages. It is convenient to be able to specify the attributes of an information structure that remain invariant during the lifetime of the structure independently of the attributes that are modified during execution. In many applications the name, number and size of the information fields remain invariant throughout the lifetime of the information structure, and it is convenient to specify these attributes by declarations when the structure is created. However, values of fields of an information structure may usually be modified during its lifetime. Such modification is usually specified by imperative program segments.

A program action which fixes attributes of an information structure is said to bind these attributes. The moment during execution at which a given set of attributes is fixed (bound) is said to be the binding time of

the given set of attributes. In performing a computation it is convenient to bind invariant attributes of an information structure at the time of creation of the structure, and to bind varying attributes of the structure at the time that values are assigned.* Declarations serve to bind invariant attributes of an information structure, while imperative program segments serve to compute varying attributes of an information structure.

The programmer sometimes has a choice as to the time of binding of certain attributes of an information structure. The following two extreme binding strategies can be used.

1. Always perform binding at the earliest possible point of time.
2. Always perform binding at the latest possible point of time.

Binding of attributes as early as possible sometimes results in more efficient program execution since it saves repeated binding of attributes. However, binding of attributes as late as possible allows the decision regarding the bound attributes to be delayed and thereby allows a greater flexibility in specifying the attribute.

Early and late binding of information structures in programming languages can be illustrated by comparing FORTRAN and ALGOL. In FORTRAN,

---

*The concept of binding is used sometimes in the sense of fixing an attribute so that it can never be changed in the future, and sometimes in the sense of merely assigning a value to an attribute irrespective of whether it can later be changed. We shall use the term in the latter sense.

storage for all information structures is normally allocated prior to the beginning of the computation. i.e. The internal representation of names is bound prior to the beginning of the computation. In ALGOL, information structures defined by declarations are created on entry to the blockhead in which the declaration occurs and are destroyed on exit from the block in which the declaration occurs.* The later binding time of ALGOL requires storage allocation for information structures to be performed during execution and may require repeated allocation of storage for a given instance of a declaration if the block is entered repeatedly.

In ALGOL there is the further option of early and late binding of formal procedures. Parameters called by value have their values bound at the time of call of the procedure while parameters called by name have only their names bound at the time of entry to the function, and are evaluated every time they are encountered during execution. FORTRAN has a form of parameter binding referred to as call by reference that is intermediate between calling by value and calling by name. Call by reference

---

*In some programming languages such as CPL[14], an ordering can be prescribed for declarations in a given block head. Some languages such as SNOBOL4[15] do not have explicit declarations, and assume that all declarative actions are executed in line just like imperative actions. Still other languages may permit both explicit declarations in block heads and in line (dynamic) declarations. Dynamic binding of declarative information provides extra flexibility at the cost of some efficiency.

results in evaluation at time of call, storage of the value local to the point of call, and binding of the location of the evaluated parameter at the time of entry to the procedure.

It is important to distinguish between cases where the time of binding does not affect the result of the computation and cases where a difference in the time of binding may affect the result. If a given attribute A is assigned value V at time T_1 by method A and at time T_2 by method B, then the two binding strategies have different results if and only if the computation changes the value V between the time T_1 and the time T_2.

Let V be the value of an ALGOL procedure parameter, let binding strategy A be call by value and let binding strategy B be call by name. Then T_1 is the time of entry to the procedure and T_2 is the time of use of the parameter. A difference in result using the two binding strategies can occur if and only if the parameter value can be changed between the time of entry to the procedure and the time of use of the procedure. A procedure which may change parameter values is said to have side effects.* If a procedure has no side effects, then the end result cannot be affected by whether a parameter is called by value or by name, but if a procedure does have side effects, then a difference in parameter binding time may affect the result.

---

*For a discussion of side effects see chapter 4 of[1].

The lambda calculus is a language with an ALGOL-like structure in which procedures effectively have no side effects.* Different binding strategies for the lambda calculus do not in general affect the function defined by a lambda expression but only the form of instantaneous descriptions I_j generated during the computation. A discussion of order of evaluation (binding) is given in section 3, and the information structure models associated with a number of different binding strategies for the lambda calculus are discussed.

The binding of different attributes of an information structure may in general occur at different points during its lifetime. It is sometimes convenient to distinguish between three different binding times for attributes of an information structure.

1. The template binding time which binds the form and attributes of a class of information structures.

2. The creation time of an instance of an information structure whose form is defined by the template.

3. The assignment time at which values of fields associated with a given information structure are assigned.

---

*In the lambda calculus procedures cannot cause side effects by changing parameter values between different times of binding. However, a difference in binding strategy may in certain pathological cases result in spurious non-termination of the computation as indicated in section 3. This side effect, which may be referred to as a side effect of control, can be avoided by ruling out binding strategies which may cause it.

In early programming languages template assignment was performed by the system and the creation time of all information structures was assumed to be prior to the beginning of the computation. In ALGOL60 structure creation for declared identifiers occurs at the time of block entry, but there are no facilities for template creation for data structures. In PL/I[16] the CONTROLLED storage allocation facility allows template definition to occur at the time of block entry, and creation of multiple instances of a structure to occur during execution. PL/I further permits creation of certain attribute fields such as array dimensions to be performed either at template binding time by BASED storage allocation or at structure creation time by CONTROLLED storage allocation. See page 317 of (1).

PL/I has CONTROLLED storage allocation facilities for data but not for programs. A function declaration in both ALGOL and PL/I defines a template for a function, and a call for a function creates an instance (activation) of the function. The data structure created by a function call is its activation record Both ALGOL60 and PL/I are carefully designed so that function calls are dynamically nested, so that the data structures associated with the activation of a function can be stored in a stack. Controlled storage allocation for activation records of functions would allow activation records for instances of functions to be created and deleted in a more flexible fashion. Some form of controlled storage allocation for

functions is required for coroutines, event notices and tasks (see section 4.10 of reference 1).

The binding of attributes of an information structure may be distributed over the template binding time, instance creation time and assignment time in a variety of ways. Some declarative languages like Euler[13], adopt the extreme strategy of having only one system template (new) which allows new names of information structures to be created. Attributes are associated with a structure not at the time of its creation, but at the time of its use. This strategy is also adopted in non-declarative languages like SNOBOL4[15], and in list processing languages. The later binding time of attributes in such languages introduces some execution time inefficiency, but allows greater flexibility in assigning and reassigning attributes of structures. One of the principal features which distinguishes list processing languages from languages for numerical computation, is precisely the later binding time of structure attributes in list processing languages.

The question of binding time is important not only in programming languages, but also in machine organization. One of the most important developments in machine organization during the last fifteen years has been to delay binding of data addresses until execution time, and to allow later and more flexible binding of information structures into storage structures. In early machines absolute addressing required data addresses to be bound prior to their use. Indexing, relocation registers and base

registers permitted successively more flexible binding of data addresses at the time of use. Paging techniques allowed later and more flexible binding of information structures into storage structures. Each of the above facilities introduces inefficiencies by delaying the binding time, but permits greater flexibility of computation.

The distinction between compilation and interpretation may be phrased in terms of the concept of binding time. Compilation results in binding of a source language program to its target language representation at translation time, while interpretation delays binding of the source language program till execution time. Here again, early binding results in greater efficiency, since it saves repeated binding, and late binding allows greater flexibility in introducing tracing and other modification at execution time.

The distinction between macros and procedures may similarly be stated in terms of binding time. When a procedure-oriented language contains macro facilities, then macro calls result in physical substitution of the macro definition in the program text prior to execution. Procedure calls result in simulated substitution of the procedure definition during execution. The earlier binding time of macros results in greater efficiency during execution. However, the delay of binding till execution time allows considerable greater flexibility during execution by delaying the binding of procedure parameters. The cost is normally only a small proportion of the total execution time of the procedure.

The above examples illustrate that the concept of binding time is very useful when distinctions between events occurring at different points of time in a computation are discussed. The present paper emphasizes the sequence of dynamic representations of a program at successive stages of execution, and the concept of binding time is an important one in this context.

The question of binding is relevant not only in the context of creation and deletion of information structures, but also in the context of specifying structural forms (templates) for classes of information structure. At this level, the question arises whether the structural form of program segments and data of a programming language should be specified (bound) at the time of language design or whether the programming language should contain facilities for specifying new structural forms suited to particular probelm areas.

Facilities within a programming language for the creation of new structural forms are referred to a definitional facilities. Designers of a general-purpose programming language cannot provide suitable program and data structures for all problem areas in which it might be used, so that general-purpose programming languages of the future will probably contain definitional facilities. However, the choice of primitive program and data structures from which non-primitive structures are to be built up still presents a problem. Languages which contain a relatively small set of

primitive program and data structures, and rely heavily on definitional facilities are sometimes referred to as core languages, while languages which attempt to provide an adequate set of primitives for general-purpose use are sometimes referred to as shell languages. Programming languages such as ALGOL and PL/I were developed as shell languages. However, a better understanding of macros allows macros to be used in developing definitional facilities. Definitional facilities have been discussed by Cheatham[17], and by Galler and Perlis[18]. Recently developed programming languages which contain definitional facilities include GPL[19] and ALGOL68[20].

2. Functions represented by Symbol Tables.

Functions with a finite domain may be represented as a set of ordered pairs called a symbol table. The first components of symbol table entries are elements of the domain and second components are corresponding elements of the range. One of the advantages of representing functions by symbol tables is that they can easily be modified. Modification of the value associated with a given element of the domain may easily be accomplished by modifying the second component of the corresponding symbol table entry. Functions with growing and shrinking domains may be implemented by allowing creation and deletion of entries in the symbol table. Sections of the domain may be temporarily pushed down and subsequently reinstated by associating a stack with the symbol table in which groups of entries may be pushed down.

Functions which are represented by symbol tables are evaluated by symbol table look up. Since the function represented by a symbol table is assumed not to depend on the order in which entries are listed, the ordering of symbol table entries may be arranged so that evaluation and modification of the function is convenient. The symbol table look-up time can be reduced from a linear to a logarithmic function of the number of entries by ordering entries lexicographically by their first components. However, such lexicographical ordering is not suited to flexible insertion and deletion of new elements. A tree structure combines the advantages of logarithmic look up with flexible insertion and deletion, at the expense of requiring extra space for linking entries into a tree structure. Hash addressing is still another way of structuring entries to combine fast accessing with flexible creation and deletion facilities.

Functions specified by symbol tables give rise to a characteristic class of information structure models. In the present section a number of examples of functions specified by symbol tables will be analyzed in order to develop insights into symbol table specification techniques in programming languages.

2.1. Encoders and Assemblers.

The simplest use of symbol tables occurs in encoding. Let S be a set of source codes and T be a set of target codes such that t_i is the target code associated with source code s_i. The symbol table consisting of the sets of pairs (s_i,t_i) determines an encoding function f which

when applied to an element $s_i \in S$ produces an element $t_i \in T$. When the encoding function is successively applied to a string of elements $s_i \in S$ it produces a string of elements $t_i \in T$.

An assembler is essentially an encoding function A whose arguments are strings of symbolic instructions. The value corresponding to a given string of symbolic instructions is an encoded string of machine language instructions. However, the encoding process in assemblers is usually implemented by more than a single symbol table. In an assembler the operation field and address fields are subject to different encoding functions. The encoding function for the operation field is usually implemented by an operation-code symbol table which is assumed to have fixed entries throughout the lifetime of the assembler.* The encoding function for the address field may be constructed during a preliminary scan of the program to be encoded. Although the function for encoding address fields is constructed during the encoding process, this function is not an example of a growing function since all name-address correspondences are assumed to be created instantaneously prior to transliteration rather than at the point at which they are encountered during execution. The effect of instantaneous creation prior to transliteration may be achieved by splitting the assembly process into <u>two passes</u> over the symbolic program, where the first pass creates the address symbol table and the second performs the transliteration.

---

*It is assumed there are no facilities for defining new operation codes.

Although assemblers are concerned principally with transliteration from symbolic to machine code, the transliteration process is controlled by the assembly process itself. Such control is established by operations called control operations. The actions associated with control operations are normally specified by a symbol table whose first components are symbolic control operations, and whose second components specify actions to be executed when the control operation is encountered. The second component of a symbol table entry corresponding to a control operation is therefore a function to be executed rather than a string to be substituted.

The symbol table entries for control operations can be stored either as part of the operation code symbol table, or as a separate symbol table. If control operations are included in the operations-code symbol table, this implies that the set of names of operation codes is disjoint from the set of names of control operations. If control operations are stored in a separate symbol table then it is usual to first consult the control operation table before consulting the operation code table. This implies that a symbolic name which is both an operation code and a control operation will always be interpreted as a control operation.

The operations performed by an assembler will be described in functional terms in order to illustrate some features of simultaneous and sequential application of functions represented by symbol tables. In function terms an assembler with control operations has an associated operation-code encoding function f_O, an address encoding function f_A and a control

operation execution function f_C. When the operation field is scanned, either f_C or f_O is to be applied depending on the content of the operation field. The composite function "either f_C or f_O" may be represented in terms of symbol table representations of f_C and f_O either by construoting a single composite symbol table representation f_{CO} or by keeping the two representations separate and attempting to apply first f_C and then f_O. If a composite representation f_{CO} is constructed then there is a restriction that the domains of f_C and f_O be disjoint. If separate representations are applied in order, there is no such restriction but an interchange in the order of application will affect the result if the domains of f_C and f_O overlap.

An application of f_O to the operation field is normally followed by an application of f_A to the address field. This can be accomplished by a state indicator set by f_O and reset by f_A to determine which function is to be used in interpreting the next field. If the function f_C rather than f_O is applicable to the operation field, then the address field has an interpretation determined by the given element of f_C.

In simple assembly programs the actions which can be performed by control operations are severely limited. However, control operations permit arbitrary actions to be specified by the symbolic code in the operation field. In a number of assemblers there are control operations for assembly time assignment and assembly time conditional transfer so that arbitrary computations may be performed at assembly time (21) (22). A further class of facilities commonly introduced into assemblers by means

of control operations are macro facilities (??) Macro facilities are an interesting example of functions defined by symbol tables and will therefore be considered in greater detail.

2.2 Macro Assemblers.

Macro assemblers allow symbol strings which constitute symbolic instruction sequences to be defined, named, and subsequently used by invoking the name. The named symbol strings are referred to as macros. Macros may have parameters. A parameterless macro associates a single string with the macro name, whereas a macro with parameters associates a class of strings with the macro name and allows selection of an element of this class by the specification of parameter values.

A macro definition specifies that a given macro name is to be associated with the symbol string that constitutes the macro body. The set of correspondences between parameterized macro names and macro bodies can be defined by a function f_M, whose domain is the set of parameterized macro names and whose range is the set of macro bodies. The function f_M is a growing function since a macro definition will add a new ordered pair to the set of ordered pairs which constitute the function definition. If the name of the newly defined macro is the same as that of a previously defined macro, the new ordered pair effectively replaces the previous ordered pair

in f_M. * If the name of the newly defined macro is a new name, this pair is merely added to the existing set of ordered pairs.

The function f_M is usually implemented by a symbol table, referred to as the macro definition table. Since macro bodies are in general of variable size, the second components of elements of f_M will be of variable size. Storage allocation for elements of the macro definition table f_M in the memory of a digital computer is usually accomplished by a linked list with each entry pointing to the "next" one. It is convenient to order entries of the linked list so that the "first" entry corresponds to the most recently defined entry, and so that links of the macro definition table are from more recently defined entries to less recently defined entries. If table look up is performed by matching macro names in a last-in-first-out order then the most recently defined instance of a given macro name is always found, and no explicit deletion of entries corresponding to redefined macro names need be performed.

The entries in a macro definition table differ from those in an operation code or address symbol table not only because they are of variable length but also because the value elements are interpreted differently during execution. Values of operation and address codes are assumed to be inert data objects which are placed in the output stream

---

*In macro systems which allow macro definitions in macro arguments (actual parameters), the convention is usually adopted that such macro definitions remain in effect only during expansion of the associated macro body. This mode of macro definition merely "pushes down" any previous macro definition having the same name.

and are not subject to any further evaluation. Values of entries in the macro definition table are assumed to be highly structured objects which must be further evaluated before being placed in the output string. Evaluation of entries in the macro definition table can be thought of as a process of scanning of the symbol string which constitutes the encoded macro body by an interpreter specially designed to evaluate macro bodies. When this interpreter encounters operation codes, address fields or control operations in the macro body, it treats them in the same way that they would have been treated by the basic interpreter for macro assembly. When formal parameter markers are encountered, they are replaced by corresponding actual parameters.

A macro call may be thought of as a symbol table look up operation in which the value resulting from symbol table look up is "executed" using a special interpreter, and using actual parameters of the macro call when the interpreter encounters formal parameters.

A macro definition is essentially a function definition. A macro call is a function call. If a macro definition has macro calls nested inside it, then a call of that macro may give rise to further function calls during execution. If a macro definition has other macro definitions nested inside it, then a call of that macro may give rise to function definitions during its execution. Since a macro definition affects the action on subsequent use of that name, the binding time of macro definitions in a macro body must be precisely specified.

Parameters of macros may in general contain macro definitions and macro calls. The binding time and manner of evaluation of macro parameters

must be precisely specified in order to completely define evaluation in macro assemblers.

An information structure model for a macro system requires two principal information components that are modified during execution.

1. A macro definition table which is augmented on macro definition and may be represented as a linked list as described above.
2. A run-time stack whose entries are macro activation records containing values of actual parameter strings and linking information. A macro call is assumed to create a new entry on the run-time stack while completion of a macro call deletes an entry from the run-time stack.

It should be noted that changes of the macro definition table by macro definitions and changes in the run-time stack by macro calls can occur during interpretation of the instruction string, during expansion of a macro body and during parameter evaluation.

A more complete discussion of the run-time structures during macro evaluation is given in (1).

In a macro assembler, macro definitions are the principal means of changing the environment of the interpreter. Macro definitions have the same role in a macro language as declarations have in a procedure oriented language, and are implemented in an essentially similar way.

2.3. Generalized Macro Systems.

A macro facility permits the definition and subsequent use of a

limited class of new structures during the execution of an assembly program. The notion of macros is, however, a good deal more general than the restricted form in which macros occur in assemblers. In macro assemblers the domain of parameters consists of one or more fields of an assembly language instruction and the range consists of a sequence of complete assembly language instructions.

There is no essential reason for restricting the domain and range of macro systems in this way, and this restriction has been relaxed in a number of macro systems (23) (24) to allow arbitrary character strings over a predefined alphabet to occur both as actual parameters and as macro values. The resulting systems are essentially string manipulation systems. However, these string manipulation systems differ from general string manipulation systems like SNOBOL4 (15) in that the format of strings which can be manipulated is restricted to the format of macro calls.

Macro systems which allow arbitrary strings to occur as macro bodies and macro parameters are referred to as macro generators. The information structure models for macro generators are essentially the same as those for macro assemblers; having a macro definition table for macro definition and a run-time stack for activation records of macro calls.

The format of a macro can be generalized to be a pattern which, when it is matched, results in substitution of a macro body for the matched pattern in the normal manner. The LIMP macro system (25) is an example of such a macro definition facility.

The patterns which constitute macros may have parameters embedded in them. Parameters in macro definition patterns may be denoted by a special marker, say * , disjoint from other symbols that may occur in the patterns. Parameters in the corresponding macro body may be denoted by a sequence of special symbols say ~1 , ~2 , ... , where ~i represents the ith parameter.

Example: Let * be a parameter marker and let "AB*CD*EF" be a macro definition pattern. Let "UV~1~2~1W" denote the macro body associated with this macro definition pattern. Let the pattern "ABPQCDREF" occur in the string being scanned. This pattern will be treated as a macro call of the above macro with a first actual parameter PQ and a second actual parameter R . The value generated by substitution of these parameters in the macro body in "UVPQRPQW" .

Macro definition patterns are sometimes referred to as templates. The sequence of fixed components of a macro template may be thought of as a generalized name whose components are separated by parameter markers, and is sometimes referred to as the distributed name of the macro.

The question of multiple definitions matching the same input pattern becomes more complex for macro templates than it was for conventional macros. The problems of ambiguity and non-deterministic input texts apply to macro systems of this kind just as they do for the pattern matching techniques used in syntactic recognition of context-free grammars (3).* A number

*See also appendix 1 of (1).

of alternative techniques for resolving local and global ambiguities will be briefly discussed.

It is convenient first to assume that the text being transformed is always scanned in a left to right order. If there are two conflicting patterns which overlap such that one starts to the left of the other, then the one which starts first is always tried first.

If two patterns start with the same symbol, then the strategy for multiple names suggested for macro assemblers can be used. i.e. The macro templates can be scanned in a last-in-first-out order so that the most recently defined pattern which matches the sequence of symbols starting at the given point will be chosen.

In syntactic analysis there is an independent criterion to determine whether templates have been matched correctly, and it may happen that use of an arbitrary criterion such as that above to determine the order of matching may not be adequate. It may be necessary to undo the result of a sequence of macro calls so that an alternative sequence can be tried. The facility is referred to as backtracking.

It is convenient to permit different domains to be associated with different parameters. Such a facility in turn requires the ability to define parameter domains. The notation of context-free grammars provides a facility for defining classes of strings that constitute parameter domains. A macro system which allows definition of parameters by use of the context-free language notation has been developed by Leaven worth (26).

A macro template with parameter types will only match a given pattern if the substrings matching the parameter markers are of the required type. This facility allows greater control over the pattern matching process.

Implicit attributes to be associated with symbols by means of declarations. A string with a given explicit syntax may be interpreted in different ways depending on the implicit attributes of the symbols.

Example: In procedure oriented languages, arithmetic expressions such as "A+B*C: are interpreted in different ways depending on whether the variable variables A , B , C are of type integer, real, complex, or matrix. The four alternative modes of interpretation can be associated with the following four macro templates: "integer + integer x integer" , "real + real x real" , "complex + complex x complex" , "matrix + matrix x matrix" . If the attributes of A , B and C are defined by declarations prior to the execution (or translation) of "A+B*C" , then the macro template with parameters of the corresponding type will automatically be chosen during expansion.

Macros whose calls are distinguished purely by implicit attributes of parameters are said to be sharing a common syntax. The sharing of a common syntax by more than one macro is referred to as syntax sharing. (see section 3.4.6 of (1)). Syntax sharing is accomplished by binding attributes of macr parameters prior to their use, and determining the applicability of a given macro definition to a given macro call by testing implicit attributes of actual parameters.

2.4 Attributes of Symbol Table Entries.

Functions represented by symbol tables permit a wide degree of variati in the organization and characteristics of the symbol table. In the exampl of assemblers and macro systems above the following aspects of symbol table organization were considered.

a) The complexity permitted in the range (second component) and in

the domain (first component) of symbol table entries.

b) The association of parameters with symbol table entries.

c) The modification, creation, deletion and pushing down of symbol table entries.

d) Different modes of accessing and interpretation (evaluation) of symbol table entries.

e) Different forms of initialization of symbol table entries.

f) Sequential, nested and multiple symbol table look-up.

Each of the above aspects of symbol table organization may be thought of as giving rise to a class of attributes which may be associated with symbol tables. The principal choices for each of the above classes of attributes will be briefly summarized.

a) The simplest symbol tables are those in which the first component is an unparameterized fixed size information component which can be directly matched, and in which the second component need not be further evaluated. Such symbol tables occur in encoding and in simple assemblers. The complexity of the second component can be increased by making it variable length, and by requiring its evaluation as in macro assemblers. The complexity of the first component can be increased by allowing first components to be a domain of patterns which require a nontrivial pattern matching procedure to be carried out in order to recognize them. A context free grammar is an example of a class of patterns whose recognition is nontrivial.

b) Parameterized symbol table entries allow each entry to represent a class of domains and to have a class of associated values. The domain of each parameter may be restricted so that a given pattern is considered to be an example of a given macro call only if parameters are in the domain defined for them. Parameterized entries may be thought of as partial instantaneous descriptions whose missing components are supplied by actual parameters.

c) The function represented by a symbol table may be changed by modification of a second component, by creation and deletion of elements, and by pushdown of groups of elements. When the function represented by the symbol table is an interpreter, then the symbol table is referred to as the environment. Creation, deletion and pushdown of symbol table entries is sometimes referred to as environment modification. The environment associated with a given interpreter may be classified into an initial environment determined by the language or system and a programmer defined environment. An alternative classification distinguishes between fixed elements of the environment whose name-value correspondence cannot be changed during execution, and variable elements which may be created, deleted and modified. A refinement of this classification is to associate attributes with each element of the environment such as fixity, modifiability, pushdownability and creatability. The facilities in ALGOL for creation and deletion of identifiers

illustrate some of the above attributes and are further discussed in Section 4.

d) The attributes of symbol table entries include the mode of access of the second component. It is important to distinguish evaluation in which the second component is treated as inert data and evaluation in which the second component is treated as a function to be evaluated.

e) When a symbol table entry is created its second component may be uninitialized, partially initialized or completely initialized. The question of initialization is essentially part of the question of value assignment. There are two principal ways of associating names with values during macro evaluation.

1. A macro definition associates a macro name with a macro body.
2. A macro call associates actual parameters of a macro call with formal parameters of the corresponding macro definition.

The correspondence between macro names and macro bodies is usually permanent in nature.* It remains in effect throughout the remainder of the computation unless a further definition with the same name occurs. The previous definition is assumed to be destroyed. A macro definition

*Except for macro definitions in actual parameter strings where the definition usually pushes down previous definitions for the duration of the macro call.

may be thought of as an assignment operation which assigns a value to the macro name in precisely the same sense that an arithmetic assignment operation assigns a value to an arithmetic variable.

The correspondence between macro parameter names and their values is assumed to be a temporary one which lastsonly for the duration of the macro expansion. While the macro is being exapnded, the name-value-correspondence determined by the parameter supersedes any previously assigned values associated with the given name. However, on completion of the expansion any previously assigned value again becomes operative, so that name-value-correspondences determined by parameters merely push down previous name value correspondences rather than destroying them. The distinction between destructive assignment and pushdown assignment is a fundamental one that will crop up both in the discussion of the lambda calculus and in the discussion of procedure oriented languages.

The mode of assignment of values to identifiers may vary in the degree to which they are initialized, and in the degree to which reassignment is permitted. The following two-way classification illustrates some of the principal choices with regard to the form of assignment.

	Single Assignment	Multiple Destructive Assignment	Pushdown Assignment
Initialized	Constants	Arithmetic Assignment	Parameter Assignment
Partially Initialized	System Functions	Macro Definitions	ALGOL Procedure Declarations
Uninitialized		FORTRAN Variables	ALGOL Arithmetic Declarations

Differences in the form of assignment are reflected in differences in the techniques for managing the symbol tables in which the name-value correspondences are represented. Destructure assignment of fixed length information items can be accomplished by using a fixed symbol table entry for successive values. Destructive assignment for variable length information items was illustrated above in connection with the macro definition table. Pushdown assignment requires stack-structured symbol tables in which entries are created and deleted in a last-in-first-out order.

The form of initialization and form of assignment are two of the attributes which must be considered in deciding how to represent the name-value correspondence for a given class of objects. A systematic analysis of the relation between symbol table structure and attributes of the objects being represented gives a great deal of insight into language and system design. This insight may be used by the programmer

to design and implement new systems more effectively.

f) There are many different ways of using symbol tables with given attributes in an interpretation process. Sequentially accessed symbol tables result in a computed value t_i when accessed with the key s_i. A nested symbol table is one in which evaluation of t_i may result in further symbol table look up. Symbol tables which implement macro definitions, syntax-directed interpretation or interpretive execution of nested program structure are examples of nested symbol tables. Nested symbol table interpreters usually require a stack to keep track of information associated with successive levels of nesting.

Multiple symbol tables arise during interpretation processes which require table look up in a number of symbol tables. The table in which look up is to be performed may be specified by a state indicator.

Many interpreters permit actions which augment push down or otherwise modify symbol table entries. The Lambda calculus and ALGOL interpreters discussed below are examples of nested symbol table interpreters which permit modification of symbol tables during execution.

3. The Lambda Calculus.

In the present section a very simple language for function representation known as the lambda calculus is introduced. The rules of transformation, rules for terminating a computation, and inherent restrictions on the order in which computations must be performed are carefully defined. A number of alternative evaluation strategies for implementing this

language as an information structure model are then considered.

The lambda calculus is a function evaluation system which was originally developed as a formal system (27). It has a facility for explicitly defining functions, but no facility for explicitly naming functions and subsequently calling named functions. The only way of establishing name-value correspondences in the lambda calculus is by associating values with formal parameters of functions. Lambda calculus evaluation models thus have no analogue of a macro definition table, and store all name-value correspondences in the analogue of a run-time stack.

One-parameter functions with parameter x are represented in the lambda calculus by λxM where M is the body of the function and may consist of an arbitrary lambda expression, and λx is the bound variable part. Since a function of the form λxM has no name (is anonymous) it cannot be explicitly called. The argument A of a lambda expression λxM must textually follow the lambda expression. An expression (λxM) denotes a lambda expression λxM followed by its argument A , which may be an arbitrary lambda expression. Evaluation of (λxMA) results in substitution of the lambda expression A for occurrences of the variable x in M .* Complete evaluation of a lambda expression is simply a chain of substitutions until no further evaluation can be performed.

---

*Certain restrictions which arise because of naming conflicts are further discussed below.

In real programming languages, evaluation of nested functions such as $f(x,g(y))$ can be thought of as a sequence of substitutions of values for functions with their arguments until no further substitutions can be performed. However, there are also other operations such as assignment of values to variables and function definition which cannot easily be specified in terms of pure substitution. The lambda calculus is a "pure" function evaluation language in the sense that all evaluation is in terms of substitution.

Substitution is not an easy concept to understand and was for many years improperly specified by logicians. One of the great contributions of the lambda calculus to mathematics was its precise and rigorous definition of the concept of substitution. In defining the lambda calculus below as a programming language we shall rigorously define the notion of substitution.

Substitution is the basis for function evaluation not only in the lambda calculus but also in real programming languages. For example a macro call or a procedure call in a programming language is essentially a two-stage substitution process.

1. Substitute actual parameters for formal parameters in the function body.
2. Substitute the function body obtained from step 1 at the current point of the program and execute it.

In implementing a programming language we do not literally perform the

two substitution steps defined above but instead create a number of structures which allow the above substitution steps to be simulated. The techniques for simulating substitution will be illustrated for the lambda calculus in order to gain an insight into alternative techniques for simulating substitution. One of the substitution techniques developed for the lambda calculus will then be applied to describe an evaluation strategy for ALGOL.

The representations of functions in purely substitutive systems like the lambda caloulus are sometimes unfamiliar, and common arithmetic operations such as + may require a large number of substitution steps. In defining a given function such as + as a lambda expression, we must pick a representation of the operation as a lambda expression and pick a representation of arguments in the domain of the operation as lambda expressions. If rep is a mapping which maps operations and arguments into a given representation as lambda expressions, then the representation of a given n-ary operation f must satisfy the condition that $rep(f)(rep(x_1),\ldots,rep(x_n)) \equiv rep(f(x_1,\ldots,x_n))$ for all n-tuples $(x_1,\ldots,x_n)$ in the domain of the function f. This condition is both a necessary and sufficient condition for consistency of a given function representation.

There may in general be many representations of a given operator and operands as lambda expressions satisfying the conditions of the previous paragraph. In (1) two very different representations of arithmetic operations over the integers are discussed. A number of real programming

languages based on the lambda calculus have been designed such as ISWIM[28] and PAL.[29] The present section is not concerned with the representation of specific functions in the language, but with the structure and implementation of the "pure" lambda calculus.

3.1 Syntax

The lambda calculus has a very simple linguistic structure when considered as a programming language. The set of permissible programs in the language can be specified by a simple set of syntactic rules, and the set of permissible transformations can be specified by a very simple set of semantic rules. Since the syntactic and semantic rules for the lambda calculus can be specified without making excessive technical demands on the reader, they will be given in detail below.

The alphabet of symbols used for specifying programs of the lambda calculus consists of a countable set V of symbols for specifying variables, and the three special symbols λ , (,) . Permitted programs in the lambda calculus will be called lambda expressions. The set of lambda expressions can be defined in terms of the basic symbols by the context free grammar $G = (N,T,S,P)$ where the set of non-terminal symbols $N = \{E\}$, the set of terminal symbols $T = (V, \lambda, (,)\}^*$, an initial non-terminal $S = E$

*The set T should be finite to meet the requirements of being a context free grammar. Since the class V of variables is countable, this requirement is not met. However, the countable set V of variables can be specified as a context free language in terms of a finite alphabet. e.g. V=(x1,x2,...). Alternatively V can be regarded as a terminal class of the context free language having the property that members of the terminal class are effectively recognizable.

and the following set of productions.

$$E \rightarrow V|\lambda VE|(EE)$$

Variables taken by themselves are lambda expressions. If E is a lambda expression then λVE is a lambda expression. If E is a lambda expression then (EE) is a lambda expression.

The following alternative definition of the set E of lambda expressions implies something about the semantics associated with components of a lambda expression.

A lambda expression consists of a variable x ; or of a bound-variable-part λx followed by a body M which is a lambda expression; or of a combination which consists of an opening parenthesis (, followed by an operator F which is a lambda expression, followed by an operand A which is a lambda expression, followed by a closing parenthesis) .

The above style of syntactic definition is due to Landin (11) and is referred to as a structure definition. A structure definition, just as a context-free language definition, defines a lambda expression in terms of a number of alternative constructions each of which consists of the concatenation of components. However, a structure definition associates suggestive names with components which hint at their semantic significance. Moreover, Landin associates operations with components that allow selection, composition and transformation of components of a structure definition.

3.2 Semantics

In order to define the rules of computation by which lambda expressions

may be transformed it is convenient to classify variables into three categories referred to respectively as binding variables, bound variables and free variables.

A variable is said to be a binding variable if it immediately follows the symbol λ.

A given instance of a variable x is said to be bound in a lambda expression M if it is a binding variable or if there is a lambda expression M' in M of the form $\lambda xM''$, where M'' includes the instance of x.

A given instance of a variable x is said to be free in a lambda expression M if it is not bound in M.

The prinoipal rule of computation in the lambda calculus is a rule for the substitution of arguments for bound variables. However, in formulating this rule, great care must be taken to avoid conflicts between names during substitution.

From a semantic point of view expressions of the form λxM represent one-argument functions which determine a rule of correspondence between arguments A and values obtained by substituting A for all free occurrences of x in M.

Example: Let λxM be $\lambda x(y\lambda x(xy))$. This expression is identical to $\lambda x(x\lambda t(ty))$ so that the identical names used for free and bound instances of x in the first representation of M is [illegible] fortuitous. If A is (uv) then the result of substitution is $((uv)\lambda x(xy))$. i.e. Inner bound occurrences of a given variable in an expression are never subject to substitution.

Substitution of arguments for values is also subject to the restriction that the substituted expression A contains no free variables that are bound in M . The reason for this restriction is illustrated by the following example.

Example: If $\lambda xM = \lambda x\lambda y(xy)$ and $A = (yz)$ then substitution of A for all free occurrence of x in M results in $\lambda y((yz)y)$, resulting in unintended identification of the free variable y of A with the bound variable y of M . Note that this can be avoided by renaming the bound occurrence of y in M before substitution.

A precise statement of the reduction rule of the lambda calculus which takes the above restrictions into account is as follows:

Reduction Rule

A lambda expression of the form (λxMA) can be replaced by the result $S^x_A M$ of substituting A for all occurrences of x in M provided M contains no inner bound occurrences of x and provided A does not contain any free variables that are bound in M .

When in an expression (λxMA) , M contains bound occurrences of x or A contains free occurrences of variables bound in M , the reduction rule cannot be applied directly. However, naming confliots can always be resolved by applying the following renaming rule for bound variables to M or to lambda expressions contained in M.

Renaming Rule

Let M be any well formed part of a lambd expression other than a

variable following the symbol λ. Then, if x is a bound variable of M, M can be replaced by the result $S^x_y M$ of substituting a variable y for all instances of x, provided M contains no free occurrences of x and y does not occur in M.

The above reduction and renaming rules completely specify the transformations which can be performed on lambda expressions. These rules effectively constitute the semantics of the lambda calculus.

The lambda calculus can be thought of as a programming language in which the primitive instructions are the renaming and reduction rules. A computation in this programming language consists of the application of a sequence of such primitive instructions. The computation is said to be complete when there is no way of renaming bound variables such that further reduction rules can be applied.

3.3 Values and Value Classes.

Reductions in the lambda calculus are analogous to evaluation steps which involve the application of a function to its arguments in other functional languages. For example in a programming language that includes arithmetic expressions, an expression such as $3 + 4 \times 5$ can be reduced to $3 + 20$ by applying $\times$ to its arguments and then to 23 by applying $+$ to its arguments. Each of these steps is analagous to a reduction, and the expression "23" does not allow further reduction steps to be applied and is said to be the value of the expression. In the discussion below the term "value" will be used to denote any lambda expression which cannot be further reduced.

Specialized function evaluation languages such as that of arithmetic expressions contain complex operations but require the arguments to be restricted to a domain such as integers or real numbers. The lambda calculus has a very simple set of operations but places no restriction on the argument domain of variables. Arguments A to be substituted for variables x can be arbitrary lambda expressions, and it can be shown that the class of lambda expressions defined above subject to the reduction and renaming rules can be used to define rules of correspondence that are isomorphic to any computable function(27).

A lambda expression to which no further reductions can be applied is said to be in reduced form. A lambda expression for which there is no finite sequence of reduction rules resulting in a reduced form is said to be irreducible. The lambda expression "$(\lambda x(xx)\lambda x(xx))$" is an example of an irreducible lambda expression. It can be shown that the general question of whether a lambda expression is irreducible is undecidable (27).

Two lambda expressions which can be converted into each other by renaming rules are said to be equivalent up to renaming. The set of all formulae which are equivalent to a given formula are said to constitute the renaming equivalence class of that formulae. For any renaming equivalence class a representative element can be chosen by imposing a lexicographic ordering on the set of all variables, and selecting names for successive bound variables according to this ordering, omitting free variable names in the given lambda expression.

Example: Let a through z be the first 26 variables of the lexicographic ordering. Then the lambda expression "λxλy((xb)y)" would have as the representative element of its renaming equivalence class the lambda expression "λaλc((ab)c)" . i.e. Names are assigned to variables in the order of their occurrence when the expression is scanned from left to right. The first bound variable x is associated with the first variable a of the lexicographic ordering. The second bound variable y is associated with the third variable c of the lexicographic ordering since the second variable b is the name of a free variable.

Since any lambda expression in a renaming equivalence class can be transformed to its representative element, and two expressions are in the same renaming equivalence class if and only if they have the same representative element, it is evident that the question of whether two lambda expressions are equivalent up to renaming is decidable.

If a lambda expression is in reduced form, then all lambda expressions which are equivalent up to renaming are also in reduced form. A renaming equivalence class of reduced form lambda expressions will be called a value class. All lambda expressions in a given value class may be thought of as representing the same value.

The value determined by a given value class is associated with all non-reduced form lambda expressions for which a reduction exists reducing them to this value class. It can be shown that the problem of determining whether two lambda expressions reduce to the same value is undecidable (27).

3.4 The Order of Evaluation of Lambda Expressions.

In the previous sections we have indicated the basic transformations which can be applied to a lambda expression, but have said nothing about the order in which the operations are applied. Indeed there is nothing in the lambda calculus which specifies the order in which reduction and renaming rules are to be applied. This is due to the fact that the lambda calculus is remarkably insensitive to the order in which transformations are performed. This insensitivity is expressed by the following theorem:

Church Rosser Theorem: For any given lambda expression all reduction sequences which terminate in a reduced form result in a reduced form in the same value class.

This theorem is proved and discussed at some length by Curry and Feys (30). It states essentially that any order of applying transformations yields the same result as any other provided that both terminate in a reduced form.

However, there are instances where one order of reduction of a lambda expression leads to a reduced form while another order of reduction is non-terminating. This situation is illustrated by the following lambda expression.

$(FA) = (\lambda x\lambda y(\lambda x(xx)\lambda x(xx))) = (\lambda xMA)$, where $M = \lambda yy$ and

$A = (\lambda x(xx)\lambda x(xx))$.

If the first reduction performed is that of substituting the argument $(\lambda x(xx)\lambda x(xx))$ for all occurrences of x in λyy , then the reduced form

λyy is obtained. If, on the other hand, an attempt is made to reduce the argument $(\lambda x(xx)\lambda x(xx))$ to reduced form before substituting it, then a non-terminating reduction sequence is obtained.

In the above example the lambda expression to be evaluated has the form (λxMA), where A is irreducible and M is reducible and contains no free occurrences of x. The lemma which follows states that these are the only circumstances in which two different evaluation sequences yield different results.

Lemma: If, for a given lambda expression, two reduction sequences exist such that one is non-terminating and the other yields a value, then the lambda expression must contain a well formed subexpression which can be reduced to a form (FA) where A is irreducible, F is of the form λxM with no occurrences of x in M, and M has a reduced form.

Outline of Proof: If the lambda expression being evaluated has a subexpression which reduces to (FA) and satisfies the conditions of the lemma, then there is clearly a non-terminating and a terminating reduction sequence. Conversely, a lambda expression for which there is a non-terminating reduction sequence must contain an irreducible lambda expression. The only way in which an irreducible lambda expression can be eliminated is by substitution as an argument into a body in which it has no occurrences.

It is desirable in evaluating lambda expressions to use an algorithm that leads to a non-terminating reduction sequence only when the lambda expression is irreducible. Fortunately, it can be shown that an algorithm

which substitutes values for bound variables in the order in which they are encountered in a left to right scan has the desired property.

Theorem: If a lambda expression is evaluated by successive evaluation of the leftmost operator-operand combination then the resulting reduction sequence will terminate if and only if the lambda expression is defined.

Outline of Proof: If evaluation of successive leftmost operator-operand combinations results in a value, then by the Church-Rosser Theorem this is the unique value of the lambda expression. It remains to be shown that if this procedure results in an infinite loop, then there is no order of evaluation that leads to a reduced lambda expression. The latter result follows from the fact that if successive reduction of the leftmost combination yields a non-terminating reduction sequence then there is no way of substituting any of the leftmost combinations so obtained for a bound variable with a zero number of occurrences. Since, by the lemma, the above condition would have to be met for a terminating reduction sequence to coexist with a non-terminating reduction sequence, it follows that if the leftmost reduction sequence is non-terminating then all reduction sequences are non-terminating and the lambda expression is irreducible.

The above theorem indicates that, for the pure lambda calculus, left to right evaluation is a correct mode of evaluation while evaluation of arguments before substitution may sometimes lead to erroneous results. Church, in his original monograph, avoids the above predicament by requiring the lambda body M of a lambda expression λxM to have at least one

free occurrence of the variable x . However, this restriction is an unnatural one and would exclude a lot of interesting lambda expressions.

If we are given a combination of the form $(\lambda x.M)A$, and if there are many free occurrences of the parameter x in M , then it clearly pays to evaluate A prior to substitution so as to save multiple evaluation of the substituted argument A during evaluation of M . For this reason a number of macro systems and other function evaluation systems insist on evaluation of arguments prior to substitution. However, if there are zero occurrences of a parameter in a function body and the parameter value is undefined, then this procedure leads to disaster.

Example: Consider the expression "if $x = 0$ then 0 else $1/x$" . If the expressions 0 and $1/x$ were always evaluated prior to testing for $x = 0$ then an undefined argument would cause an error when $x = 0$. However, this expression avoids evaluating $1/x$ when $x = 0$ so that a proper order of evaluation always yields a defined result.

The notions of evaluating arguments before and after substitution have their counterparts in ALGOL. Evaluation of an argument prior to substitution corresponds to a call by value while substitution of the unevaluated argument corresponds to a call by name. Here again a call by value will save multiple evaluation if the argument is used repeatedly, but will lead to a spurious error if it corresponds to an undefined argument which is never used.

In the case of procedure oriented language like ALGOL, the situation

is further complicated by the fact that the Church-Rosser theorem does not hold. In a procedure-oriented language, evaluation of a function $f(x)$ may cause side effects in the sense that it may modify values of quantities that are accessible to other function modules. Side effects cause the process of evaluation to become highly sensitive to the order in which operations and function modules are applied to their operands. Procedure-oriented languages accordingly have built into the language explicit sequencing rules for the evaluation of operator-operand combinations.

3.5 Lambda Calculus Machines.

An information structure model (I,F) associated with the lambda calculus has a component I which consists of the set of representations of lambda expressions, and a component F which consists of a specification of the renaming and reduction rules in terms of the transformations they perform. The closure $cl(I)$ depends on the specific evaluation strategy used for implementing renaming and reduction rules.

The set I of initial representations can be specified as $E \times (1)$ where E is the set of representations of lambda expressions, and (1) specifies that the next instruction pointer points to the first character of E. Figure 4, gives the initial representation of the lambda expression $(\lambda x\lambda y(x(xy))\lambda u\lambda v(u(u(uv))))$. In (1) this lambda expression is shown to be a representation of $(2\ 3) \equiv 3^2$.

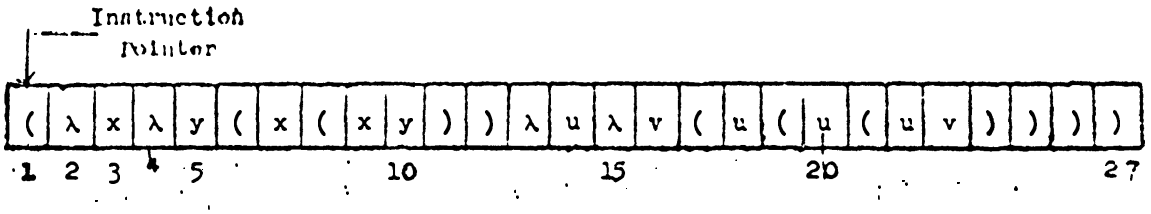

Figure 4. An initial representation of a lambda expression.

Computations are assumed to proceed by scanning the sequence of characters pointed to by the instruction pointer, and performing a transformation of the information structure determined by the recognized character. Each transformational step may be defined by a conditional expression of the form "if p_1 then A_1 else if p_2 then A_2 ... else if p_n then A_n" where each p_i, $i = 1,2,\ldots,n$, is a predicate testing for the presence of a character or character class at the current point of execution and A_i is the action to be taken when the character determined by p_i is recognized.

The reduction and renaming rules are global transformation rules with no specification regarding the order in which they are to be applied. The class of execution algorithms of the previous paragraph specify an ordered sequence of primitive transformation steps which are of a totally different nature from the reduction and renaming rules. The domain of each primitive transformation step is restricted to a single character, making primitive transformation steps similar in form to instructions on a digital computer. It is appropriate to call information structure models having primitive instructions of this kind lambda calculus machines.

Four different lambda calculus machines will be described, corresponding to different strategies of simulating the substitutions specified by the reduction and renaming rules.

1. The literal substitution machine in which the reduction rule is implemented by literally substituting arguments A for bound variables x when a binding expression λx is encountered during execution.
2. The pointer substitution machines in which pointers to arguments are substituted at binding time and arguments are substituted for pointers when the pointer is encountered in the body of the lambda expression.
3. The LISP machine in which correspondences between bound variables and their values are stored in a symbol table (pair list) and substitution of arguments for bound variables is performed when they are encountered in the body of a lambda expression.
4. The fixed program machine in which correspondences between bound variables and their values are stored in a symbol table as in the LISP machine. When a bound variable is encountered in the body of a lambda expression control is transferred to the corresponding argument by means of a "procedure call", the argument is executed, and control is then returned to the point of call. In the fixed program machine the lambda expression is never modified but information structures must be created during execution to

store name-value correspondences and linkage information.

The differences between the above models is essentially a difference in the form of binding both at binding time during execution of a binding expression λx , and at use time when a bound variable is encountered in the body of a lambda expression. The literal substitution and pointer substitution binding strategies are rarely used in implementing real programming languages. The strategy of the LISP machine is used in LISP (as defined by the LISP APPLY function)[10] and in macro systems. The strategy of the fixed program machine is used in implementing procedure calls in procedure oriented languages.

For each of the above models it is necessary to distinguish between a passive binding variable x occurring in a context λxM in which no argument A is available, and an <u>active</u> binding variable x occurring in a context (λxMA) in which there is an argument A to be substituted for occurrences of x in M . This can be accomplished by a <u>parenthesis-count register</u> P which holds the number of parentheses encountered prior to the occurrence of the binding expression. If $P = 0$ the binding variable is taken to be passive while if $P > 0$ the binding variable is taken to be active.

In the literal substitution machine the closure $cl(I)$ of I is $E \times P \times \{1\} \times \phi$ where P is the set of integers in the parenthesis count register, and ϕ is an output component which will contain the evaluated lambda expression on termination.

The set of operations in the literal substitutions can be specified as follows.

1. If (then P = P + 1
2. else if λx and P > 0 then perform substitution and possible renaming and set P = P - 1 . Throw away closing parenthesis.
3. else if λx and P = 0 then output λx .
4. else if x then output P left parentheses followed by x and set P to zero.
5. else if) then output) .

Line 2 of the conditional expression above involves a very complex transformation of the information structure E which inoludes testing for possible renaming, and substitution of variable length arguments for bound variables in the string E . Characters of the new string will have to be renumbered because of the variable length substitutions.

The pointer substitution machine avoids both substitution of variable length arguments in the program string, and tests for conflicts of names between bound variables. In the pointer substitution machine, the reduction rule for active binding variables x is implemented by substitution of fixed size pointers p_i to the argument A for all free occurrences of x in M . When such a pointer is encountered during scanning of the body M , the pointer is replaced by the literal argument A , and the instruction pointer is moved to point to the first character of A .

Example: Consider evaluation of $(\lambda x(xx)\lambda y y)$ by the pointer substitution machine. The opening parenthesis sets the parenthesis-count register P to 1. The binding expression λx results in replacement of both instances of x in the body $M = (xx)$ by p_1 resulting in an expression $(p_1 p_1)\lambda y y)$ with p_1 pointing to $\lambda y y$. The parenthesis is then scanned setting P to 1. Scanning of the pointer p_1 then results in the expression $\lambda y y p_1)\lambda y)$ and binding of y results in $p_2 p_1)\lambda y y$ with p_2 pointing to p_1 and with P = 0. Evaluation of p_2 results in replacement by $\lambda y y$. Since P is now zero, λy is interpreted as a passive binding expression and output. y is then output and a mechanism (not here described) recognizes that execution of the lambda expression has been completed.

A complete definition of a pointer substitution machine together with worked examples is given in (1).*

The instantaneous descriptions I_j of a pointer substitution model may be transformed during execution of a binding expression by substitution of pointers for variables without changing the size of the information structure. When a pointer is encountered during execution, a variable length argument may replace the fixed length pointer. However, substitution of a variable length string for a fixed length pointer may occur only when

---

*In (1) the pointer substitution machine is called the basic lambda calculus machine.

there are no further constituents to the left of the pointer. This effectively makes the information string that initially contains the lambda expression into a stack which may grow at the point corresponding to the currently executed instruction. Internal information fields of the stack may be modified without changing their size. This stack is referred to as the workstack in Reference 1.

The literal and pointer substitution machines require modification of the information string which constitutes the program both at binding time and at use time. The LISP machine is an information structure model which requires no modification of the program component at binding time but does require modification of the program component at use time. The fixed program machine is an information structure model in which the program component remains unmodified both at binding time and at use time.

In the LISP machine, execution of an active binding expression λx results in the creation of a symbol table entry (x,p_A) where p_A is a pointer to the argument A associated with x. When a variable x is encountered in the body M of a lAmbda expression, then the symbol table is consulted to see if x has been defined. If so, the the argument A corresponding to the pointer p_A Associated with x is substituted for x in the program component. Otherwise the variable x is assumed to be a free variable.

The substitution performed at symbol use time in a LISP machine is essentially the same as the substitution performed at use time in a pointer

substitution model. However, the program component of a LISP machine is a "pure" stack which permits last-in-first-out creation and deletion at the point corresponding to the currently executed instruction, and requires no internal reading or writing. This simplification of the stack component is accomplished at the expense of creating an auxiliary symbol table component.

Multiple entries having the same name (first component) may occur in the symbol table of a LISP machine. Proper handling of symbol table entries so that name conflicts are always properly resolved requires some care and is discussed in (1).

The information structure model for the LISP machine is discussed in (1), has been explored by McCarthy (10), and will not further be discussed here. However, it is significant that from our present point of view the LISP machine is regarded merely as one of a number of alternative models of evaluation. The model of evaluation that is closest to that used in procedure oriented languages is the fixed program model.

In the fixed program machine evaluation of an active binding expression results in a symbol table entry just as in the LISP machine. When a bound variable is encountered in the body of a lambda expression, control is transferred to the associated argument as though it were a closed subroutine. When transferring control, linkage information must be stored to allow return to the point of call when execution of the argument string has been completed.

The fixed program of a fixed program machine consists of a sequence of

nested function modules which, during execution, can be executed recursively to an arbitrary depth.

Example: $(\lambda x((xx)x)\lambda x((xx)x))$ is an irreducible lambda expression which during execution results in progressively deeper recursive reentry of the first component.

Although the dynamic level of nesting during execution may become arbitrarily deep, the number of different bound variables which are accessible at a given point of execution is independent of the dynamic depth of nesting and depends only on the number of static one-parameter functions in which it is embedded.

The level of nesting of a given symbol in a lambda body is defined as the number of nested functions of the form λxM which enclose the symbol. When executing a program at a depth of nesting n there are at most n bound variable x_i whose values are accessible at that point of execution. The set of accessible values are those which correspond to the most recently executed instances of the n binding expression λx_i which enclose the current point of execution in the static program.

The lambda body associated with a given binding expression λx is said to be the scope of the bound variable x of that binding expression. Symbols of the lambda body at nesting level n are said to be within the scope of precisely n bound variables.

A procedure call within the scope of a given set of bound variables may give rise to a chain of procedure calls. Succcessive procedure calls of

such a chain are said to be dynamically nested in preceding calls are sometimes referred to as inner procedure calls. Exit from a chain of procedure calls is always in a last-in-first-out order.

The set of accessible bound variables at the current point of execution will be called the environment at that point of execution. The number of accessible elements at a given point of execution of the program is an invariant of the initial representation of the program. We will take advantage of this fact in managing and updating the environment.

The environment undergoes modification at the following points during execution of a program on the fixed program machine:

1. When a binding expression λx is scanned, a new entry is added to the environment. This entry has the form (x,A) if there is an argument which corresponds to λx and has the form (x,b_i) if this binding expression has no associated operand.

2. When evaluation of a given function module is completed, the top entry of the environment is deleted. i.e. The point of execution moves from level of nesting n to a level of nesting $n-1$ and the number of entries in the environment is decreased by 1.

3. When transferring control to a bound variable x at nesting level n which is defined by a binding expression at level $k < n$ the $n-k$ environment elements associated with nesting levels $k+i$, $i=1,2,\ldots,n-k$ must be temporarily pushed down since they are not accessible in scanning the argument string associated with x. If the argument of

x is itself a lambda expression with internal levels of nesting, new environment elements k+1,k+2,... will be created during execution of the argument-expression. However, these environment elements will be deleted before control is returned to the point of call.

4. On return from execution of an argument expression at level k to a point of call at level n > k , the n-k environment elements pushed down during execution of the argument expression must be restored.

Instead of literally pushing down and restoring name-value pairs during the execution of the argument expression associated with a bound variable, it is convenient to store all name-value correspondences in a symbol table whose entries are created in a last-in-first-out order and to store the current environment at a given point of execution as a vector of pointers to symbol table entries.

The symbol table of name-value pairs is a stack with respect to creation and deletion, but allows arbitrarily deep read-only references within the stack by means of environment pointers. This stack will be referred to as the symbol table stack. The symbol table stack has as many entries as the dynamic level of nesting during execution and may therefore grow to arbitrary size.

In addition to the symbol table stack a stack is needed to remember return links for return to the point in the source program following the "procedure call" when evaluation of the argument-expression associated with a bound variable is completed, and a stack is needed for pushing down sets

of n-k environment pointers when execution of an argument expression is initiated. These two stacks can be combined into a single stack each of whose entries contains a variable number n-k+1 of elements, where n-k elements are pushed-down environment pointers and the remaining element is the return link. A stack of this form will be referred to as a procedure stack.

A symbol table stack entry created on execution of a given binding expression remains in existence during all chains of procedure calls originating within the scope of the binding expressions, and is deleted when execution of the body which constitutes the scope is completed. A procedure stack entry created on entry to a given procedure remains in existence throughout the lifetime of all symbol table stack entries of inner (dynamically nested) function modules, and no environment stack entries created prior to a given procedure stack entry can be deleted before deletion of the procedure stack entry. It follows that procedure stack entries and symbol table stack entries can be stored in a single stack which we will refer to as the run-time stack.

The run-time stack contains three kinds of entries:

1. Entries which specify name-value correspondences, are created on execution of a binding expression, and are deleted when execution of the body which constitutes the scope of the binding expression is completed.

2. Entries which specify return links, are created on call of an

argument-expression, and are deleted when execution of the argument-expression is completed.

3. Entries which specify environment pointers and are created on entry to and deleted on exit from argument evaluation.

Creation and deletion of information structures in the fixed program machine is associated with entry to and exit from function modules. The unit of information created on entry to a function module and deleted on exit from a function module will be called an activation record. An activation in general has the following format.

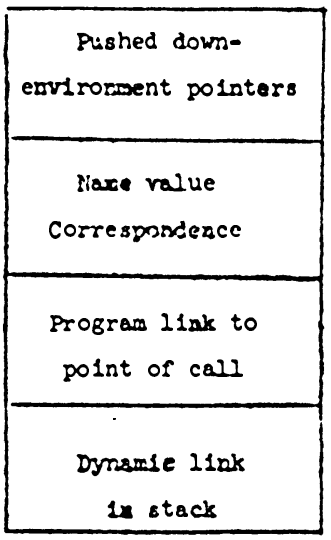

Figure 5. Activation Record with Environment Pointers

Instead of pushing down environment pointers on call of an argument, the sequence of activation records in the currently accessible environment can be linked by a sequence of static links referred to as a static chain.

This results in activation records having the following format.

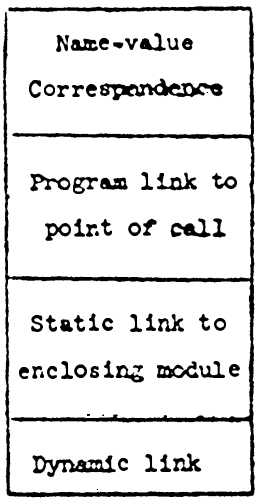

Figure 6. Activation record with Static Chains.

The number of activation records in the run-time stack depends on the dynamic level of nesting at a given point of execution. The maximum level of nesting to which a lambda expression will give rise during execution cannot in general be determined from the static program.

The number of environment levels in which a given point of the program is enclosed during execution is a function of the static program. Thus, the number of links in the static chain is always invariant.

The above model was developed in some detail because it forms a basis for evaluation strategies in procedure oriented languages. In the next section a model similar to that of the fixed program machine will be used to describe ALGOL evaluation.

4. Information structure Models for Procedure Oriented Languages.

4.1 Function Modules in ALGOL.

ALGOL is a function evaluation language whose function module structure is similar to that of the lambda calculus. Its basic building blocks are function modules f (blocks and procedures) which determine a rule of correspondence $I_B \overset{f}{} I_A$ where I_B is an instantaneous description prior to execution of the function module and I_A is the instantaneous description resulting from the application of f to I_B.

It will be assumed that the information structure model for ALGOL has a fixed program component which consists of the sequence of characters that constitute the program or some compiled version of the program string. During execution information structures must be created to keep track of name-value correspondences in the environment, and to store linkage information. The information structure created on entry to a function module and deleted on exit from a function module will be called an <u>activation record</u>.

The function modules of ALGOL may be nested just as in the lambda calculus. A complete ALGOL program consists of a function module called a <u>block</u> which may have other blocks nested inside it to an arbitrary level. Blocks are always entered and exited in a last-in-first-out order, and their activation records can therefore be stored in a stack. When the program string corresponding to a nested set of blocks is executed, an activation record is created for each block as it is entered, and is deleted

on exit from the block, as illustrated in Figure 7.

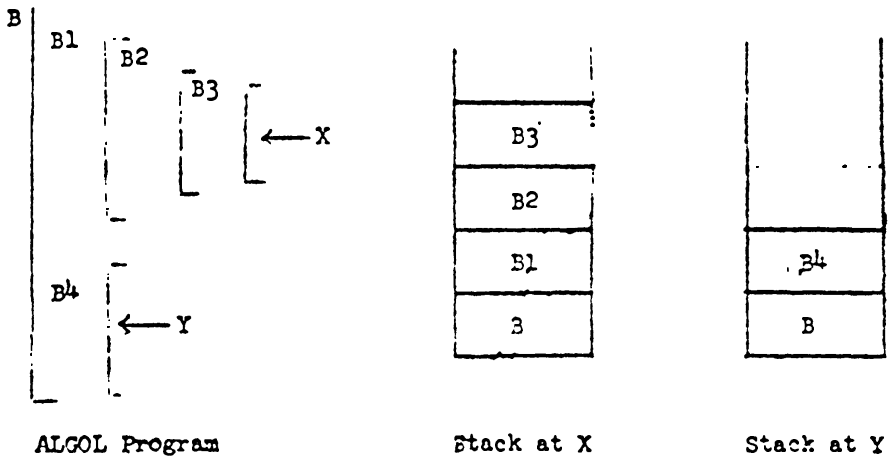

Figure 7. Activation Records for ALGOL Blocks.

Figure 7, illustrates an ALGOL program with five blocks B , B1 , B2 , B3 , B4 with the indicated nested structure. On entering the program, an activation record for B is created. During linear execution of this program, activation records for B1 , B2 , B3 are successively created and then deleted in a last-in-first-out order. An activation record for B4 is then created and deleted. Finally the activation record for B is deleted and execution of the program terminates. The activation record stack at points of execution X and Y is indicated in Figure 7.

An ALGOL block may have information structures declared in its block-head. Such information structures are created on entry to the block and deleted on exit from the block. The information structures declared in a

blockhead may be of the type integer, real, Boolean, array, procedure, label* and switch. Each type defines a class of information structures, and is subject to a class of operations of the programming language for objects of that type. Primitive types of a programming language may be regarded as system template specifications, and declarations may be regarded as commands to create an instance of the type specified by the template and to associate a specified name with the created instance.

Declarations of integer, real and Boolean information structures are uninitialized declarations, and may have values defined and redefined by assignment statements. Arrays of integer, real and Boolean information structures are likewise uninitialized declarations. Procedure declarations are partially initialized declarations. Label and switch declarations are initialized declarations.

An ALGOL block consists of the symbol begin followed by a sequence of declarations, followed by a sequence of statements, followed by the symbol end. A block is structurally similar to a lambda expression (λxMA), particularly when the lambda expression is rewritten $(\lambda(x,A)M)$. However, there are the following differences.

1. All bound variables in lambda expressions are of the same type while declared quantities in ALGOL may have one of a number of different

*Information structures of the type label cannot be explicitly declared in a blockhead. However, an occurrence of a labelled statement implies a declaration of that label in the immediately enclosing block.

types. The type associated with bound variables of lambda expressions effectively corresponds to the ALGOL type "parameterless procedure."

2. In the lambda calculus values can be associated with bound variables only by explicitly writing them after the lambda body. Declared quantities in ALGOL may have their values specified by assignment, by associating values with parameters during procedure calls, and in a number of other ways.

An ALGOL procedure declaration is a function module which has many similarities of structure with ALGOL blocks and differs principally in the mode of entry and exit. Both blocks and procedures consist of an initial portion called a function heading which specifies bound variables of the function module, and a function body which specifies the sequence of actions during execution of the function module. During execution, the function heading causes actions of creation and deletion while the function body is associated with actions of transformation.

The mode of entry to an exit from block and procedure function modules is different. Blocks are entered when control reaches the begin symbol during statement execution, and result in execution of the textually following statement on exit through the end statement. Procedures are entered when a procedure call is executed in a function body enclosed by the block in which the procedure is declared, and return control to the point of call on exit from the procedure. In this respect procedure calls are just like bound variables which occur in the body of a lambda expression. However,

procedure calls are more complex forms of transfer of control than bound variables of lambda expressions. A variable in the body of a lambda expression is essentially a parameterless procedure call.

The form of binding of bound variables differs for blocks and for procedures. In establishing a name-value correspondence for a declaration in a block head, both the name and the value component are assumed to be created at the time of declaration. In establishing the name-value correspondence for a procedure parameter, the value component is already in existence and the name component of the procedure must be linked to an existing value component. The different information structure transformations implied by these two forms of linking are futher discussed below.

There are two different ways in ALGOL of binding the value of a procedure parameter referred to respectively as call by value and call by name. Call by value corresponds to evaluation of the argument prior to entry into the procedure and binding of a copy of the evaluated argument at the time of entry to the procedure. Call by name corresponds to evaluation of the parameter in the environment of the calling program every time its value is required during execution of the procedure. Every execution of a parameter called by name is effectively a parameterless procedure call to a procedure defined at the point of call for evaluating that parameter.

4.2 Environment Management in ALGOL.

The set of all symbols that are meaningful to the interpreter at a given point of the computation is called the environment at that point of

the computation. Initially the set of all symbols that are meaningful to an ALGOL interpreter are system symbols such as begin , for, + , real , etc., constants such as 11.65 , true, false, etc., and a set of initially defined identifiers such as sin, cos, etc.

During execution the set of symbols meaningful to the ALGOL interpreter can be augmented by declarations. The environment may be augmented by a set of identifiers on entry to a function module. On exit from the function module the corresponding set of identifiers is deleted from the environment. When the function module is a block, the identifiers are specified by declarations. When the function module is a procedure, the identifiers are specified by parameter specifications. Since entry to and exit from function modules is always in a last-in-first-out order independently of whether the function modules are blocks or procedures, activation records of function modules can be stored in a stack.

The level of nesting of a given symbol of the static program will be defined as the sum of the number of block and procedure heads in which it is embeded. Each block and procedure may have a group of bound variables declared in its head. When executing a symbol at nesting level n , the set of accessible name-value correspondences consist of the n groups of bound variables associated with the most recent instance of execution at each level of nesting.

The set of n groups of bound variables accessible at nesting level n are stored in n activation records, which need not necessarily be

segment in the procedure stack. Access to these n activation records can be accomplished by a set of n pointers called a current environment vector pointing to the base of each of the n accessible activation records. This is accomplished by a chain of links called a static chain linking each activation record to the most recent activation of the next statically enclosing function module. An information structure model which uses a current environment vector to access activation records is called a current environment vector model while an information structure model which uses a static chain to access activation records is called a static chain model. The current environment vector model permits more rapid access to activation records than the static chain model, but requires greater overhead on entry to and exit from procedures, as indicated in (1).

The efficiency of execution of an ALGOL interpreter can be increased by a compilation phase which replaces all identifiers by integer pairs (i,j) where i is the static nesting level of the declaration for that identifier and j is the relative address within its activation record at which the value of the identifier will be stored. The integers i and j for any given occurrence of any identifier can easily be determined at compile time. When identifiers are represented by integer pairs (i,j), then the value of an identifier in the current environment vector model can be accessed by indirect addressing through the ith entry of the current environment vector and then taking relative address j. In the static chain model it is convenient to represent identifiers by (r,j) where r is the

difference in nesting level between the point of use of the identifier and the point of declaration. The integer r specifies the number of links along the static chain which must be followed in order to reach the activation record in which the identifier is defined.

The differences between the current environment vector model and the static chain model is illustrated by the following example.

```
B1: begin real a;
      real procedure P(x);
        B3: begin real c;
X------------------>
            end
        B2: begin real b;
              a := P(a+b);
            end
    end
```

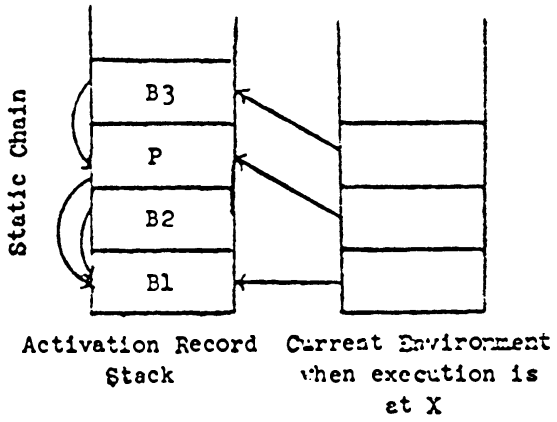

Figure 8. Comparison of Current Environment and Static Chain Models.

In this example, control enters the block B1 and then enters the block B2 . Execution of the statement a := P(b); results in a call of the procedure P and execution of the block B3 . When executing the block B3 , the bound variables of the enclosing function modules B3 , P and B1

are directly accessible, but bound variables in the function module B2 [illegible] are not directly accessible. This is reflected in the current environment vector by the fact that it contains only pointers to the three accessible blocks B3 , P and B1 and is reflected in the static chain by the fact that B2 is not in the sequence of activation records pointed to by the static chain.

Execution of a parameter called by name corresponds to a parameterless procedure call to the environment at the point of call. Since a procedure call gives rise to an activation record during its execution, a parameter call by name likewise given rise to an activation record. The activation record for a parameter called by name contains linkage structures but no parameter specification structures. In the current environment vector model, the current environment vector during evaluation of the parameter would be the current environment vector at the point of call. In the static chain model, the static chain link of the parameter called by name must receive special treatment since it is not counted as one of the r links to be followed when an identifier (r,j) is encountered.* The activation

---

*An extra nesting level can be automatically associated with actual parameter expressions by creating explicit parameterless procedures for these expressions in the innermost block in which they occur, and replacing the expression by a call of the procedure. Thus a call P(a+b) would cause a procedure, say [illegible] body a+b to be created at compile time. P(a+b) [illegible]. In a language which does not have call by [illegible] be used to simulate call by name.

record for the parameter called by name is effectively an extension of the activation record at the point of call for purposes of accessing. However, when control is in the activation record because of a call by name, control is returned on exit to the called program, while normal termination of execution returns control to the calling program.

A parameter call by name may be thought of as a recursive activation of the calling procedure which uses the same copy of the locally declared bound variables as its parent, but which requires different linkage information. It is implemented by effectively pushing down the linkage portion of the activation record at the point of call while making arrangements to share values of locally declared bound variables with its parent.

The first implementation of ALGOL along these lines was developed by Dijkstra[31] and described in detail by Randell and Russell[32]. This paper provides a framework for describing and comparing implementations of programming languages.

A first approximation to an information structure model for ALGOL can be specified in terms of the six components (P,W,S,I,O,X) where P is the program component, W is the stateword component, S is the stack component, I is the input component, O is the output component and X is the own variable component. Instantaneous descriptions in ALGOL can be denoted by quintuples (P,W,S,I,O,X) and execution of instructions can be specified in terms of how each of the components of the instantaneous description are modified.

Each of the components has a characteristic substructure and mode of access. Thus the P component is pure procedure and is accessed through W in a read-only mode. The W component contains a pointer to the current point of execution, to the base of the top activation record of the stack, and may contain the current environment vector in the current environment vector model. In a general model for ALGOL, the W component is not of fixed size, since the program size, stack size and current environment vector size are unbounded. The program pointer and current environment vector can be stored in a linearly bounded information component in the sense of linear bounded automata. However, the stack size is not a predictable function of the size of the program, so that there must in general be provisions for more than linearly bounded growth of the stateword. In practice, a restriction is usually placed on program and stack size, and the stateword W is of fixed size. Programs and stacks larger than this fixed size may be handled by overlay techniques.

The stack component S is the component which contains data, linkage, and accessing information during execution. It is augmented by an activation record on entry to a function module and diminished by an activation record on exit from a function module. The top of the stack S may be used also for storage of temporary quantities during expression evaluation. If the stack is organized as described above, then access to the interior of the stack is permitted either through the current environment vector or through the static chain. The mode of access to the interior of the stack

may be read-mode access, or write-mode access when assigning the value of a variable. However, write-mode access to the interior of the stack can change only the value of an information component and not its structure.*

The input component is assumed to be a read-only one-way device like the input tape of a finite automaton. It is assumed to be accessed by a system function "read" which is not part of "pure" ALGOL. The output component is a write-only one-way component which is assumed to be accessed by a system function "write".

The component X is a storage structure for own variables. own variables permit information structures to be created which are private to the block in which they are created but which continue to exist between activations of the block. There is some confusion over own information structures whose size may vary or which may be duplicated by recursive activations of the module in which they are defined. The precise organization of the information component X depends on the way in which this confusion is resolved.

An information structure model for ALGOL can be specified as a conditional expression of the form "if P1 then A1 else if P2 then A2 ... else if PN then AN"; where P1 , P2 ,..., PN are the possible

---

*Since the precision of primitive data types is not fixed in ALGOL, it is, in principal, necessary to associate variable length information structures with primitive data types such as integers. Assignment of a new value might then require a change in the size of the structure holding the value. It is here assumed that primitive data structures have a fixed precision and are stored as fixed size information structures.

patterns which may occur as domains of primitive instructions, and A1 , A2 ,..., An are information structure transformations triggered by the respective patterns. In order to keep the number N of patterns down to a manageable size, the patterns would normally be parameterized. Actions A might also be conditional expressions of the same form as the expression in which they are embedded. This results in a tree structure of parameterized pattern tests.

The transformations A can be specified in terms of transformation of P,W,S,I,O,X . A complete and rigorous specification of ALGOL as an information structure model would require a great deal of effort. However, a modelling language with appropriate transformation primitives would facilitate the task of specifying information structure models for languages such as ALGOL.

Different evaluation strategies result in different information structure models. For example the current environment and static chain models give rise to two different information structure models for ALGOL60 interpretation. Still another model developed by Huxtable[34] and described in section 4.7 of (1), associates activation records with recursive program levels instead of with blocks and procedures. Execution-time efficiency of this model is increased by earlier binding time of activation records associated with function modules.

4.3 Models for Programming Languages.

ALGOL is a very simple language compared with PL/I[16], ALGOL 68[20], GPL[19], and other recently proposed procedure oriented languages. However, the nested function module structure of ALGOL carries over to the above languages. The information structure model for ALGOL can serve as a starting point for developing models of more complex languages with a nested function module structure. Some characteristics of information structure models for PL/I will briefly be illustrated.

PL/I programs have a basic block structure like ALGOL. This means that entry to and exit from function modules is in a last-in-first-out order, and activation records of function modules can be stored in a stack. PL/I differs from ALGOL principally in having a larger set of primitive data types, in allowing more varied attributes to be associated with data types, and in permitting more flexible creation and deletion of information structures than is permitted in ALGOL.

The greater complexity of PL/I can conveniently be described by indicating the kinds of structures to which PL/I source language features give rise during execution. Many of the structures can be described in terms of generalizations of structures that arise in ALGOL or in terms of structures that arise in models of list processing languages.

For example PL/I structure declarations can be thought of as generalized arrays. The structure specification may be implemented as a generalized dope vector. Accessing of structure components can be described as a

generalization of array accessing.

The unpredictable creation and deletion resulting from CONTROLLED and BASED storage attributes can be characterized in terms of the dynamic storage allocation strategies required to implement them.

A detailed definition of PL/I as an information structure model requires a great deal of effort and is being undertaken by IBM Vienna[35]. However, insights into the structure of PL/I can be obtained with relatively little technical effort by considering only higher level information components. A similar effort[36] undertaken for SNOBOL4[15], which is a very rich language having a totally different structure than PL/I, has shown that this approach is not restricted to ALGOL-like languages.

Conclusions

An entity like a programming language may appear to be complex and totally incomprehensible if the wrong tools are used in describing it. However, the use of an appropriate conceptual framework may allow the same language to be simply described. It is felt that information structure models provide a uniform method of simple description of programming languages. The uniformity of approach facilitates comparisons both among programming languages and between programming languages and the structures that arise in automata theory and logic.

Computer science is concerned with the representation and transformation of information structures. Information structure models provide a tool

for modelling representation and transformation. Functions are one of the principal kinds of objects that programmers wish to represent. The study of representation and transformation of functions is therefore central to programming. This paper has tried to bring together a number of concepts relating to the representation and transformation of functions. It is felt that the explicit study of function representation along the lines suggested in this paper is an area with great research potential, and that information structure models may provide a key to the development of a systematic discipline of programming.

REFERENCES

1. P. Wegner, Programming Languages, Information Structures and Machine Organization, McGraw Hill, 1968.

2. P. Naur, et. al., "Revised Report on the Algorithmic Language ALGOL 60," Communications of the ACM, January 1963.

3. S. Ginsburg, The Mathematical Theory of Context Free Languages, McGraw Hill, 1966.

4. J. McCarthy, "Towards a Mathematical Science of Computation," Proceedings of the IFIP Conference, August 1962.

5. R. M. McClure, "TMG - A Syntax Directed Compiler", Proceeding of the 20th ACM conference, Cleveland, August 1965.

6. J. A. Feldman, "A Formal Semantics for Compiler Languages and its Application to a Compiler Compiler", Communications of the ACM, January 1966.

7. J. C. Reynolds, "An Introduction to the COGENT Programming System", Proceedings of the 20th ACM Nat'l. Conference, 1965.

8. R. A. Brooker and D. Morris, "A General Translation Program for Phrase Structure Grammars," JACM, January 1962.

9. M. Davis, Computability and Unsolvability, McGraw Hill, 1958.

10. J. McCarthy, et. al., LISP 1.5 Programmer Manual, MIT Press, 1962.

11. P. J. Landin, "The Mechanical Evaluation of Expressions", The Computer Journal, January 1964.

12. R. W. Floyd, "A Descriptive Language for Symbol Manipulation", JACM, October 1961.

13. N. Wirth and H. Weber, "Euler - A Generalization of ALGOL and its Formal Definition, Comm. ACM, January and February 1965.

14. D. Farren, et. al., "The Main Features of CPL", Computer Journal, July 1964.

15. R. E. Griswold, J. F. Poage, and I. P. Polonsky, "Preliminary Report on the SNOBOL4 Programming Language", Bell Telephone Laboratories, Holmdel, New Jersey, November 22, 1967.

16. PL/I Language Specification, IBM Systems Reference Library, File 5360-29 Form C28-6571-4.

17. T. E. Cheatham and K. Sattley, "The Introduction of Definitional Facilities into Higher Level Programming Languages", Proc. Fall Joint Computer Conference, San Francisco, November 1966.

18. B. A. Galler and A. J. Perlis, "A Proposal for Definitions in ALGOL", Comm. ACM, April 1967.

19. J. V. Garwick, J. R. Bell and L. O Krider, "The GPL Language", Programming Technical Report TER-D5, Control Data Corporation.

20. A. Van Wijngaarden, Editor, "Draft Report of the Algorithmic Language ALGOL 68," Supplement to ALGOL Bulletin No. 26, January 1968.

21. P. Wegner, Introduction to Symbolic Programming, Griffin, 1963.

22. M. D. McIlroy, "Macro Instruction Extensions to Compiler Languages", Comm. ACM, April 1960.

23. C. N. Mooers, "TRAC - A Procedure Describing Language for the Reactive Typewriter", Comm. ACM, March 1966.

24. C. Strachey, "A General Purpose Macro Generator", The Computer Journal, October 1965.

25. W. M. Waite, "A Language Independent Macro Processor", Comm. ACM, July 1967.

26. B. Leavenworth, "Syntax Macros and Extended Translation", Comm. ACM, November 1966.

27. A. Church, The Calculi of Lambda Conversion, Princeton University Press, 1966.

28. P. J. Landin, "The Next 700 Programming Languages", Comm. ACM, March 1966.

29. A. Evans, "PAL - Preliminary Description", MIT, February 1968.

30. H. B. Curry and R. Feys, Combinatory Logic, North Holland, 1968.

31. E. W. Dijkstra, "An ALGOL 60 Translator for the X1 Computer", ALGOL Bulletin Supplement Number 10, 1960.

32. B. A. Randell and L. J. Russell, ALGOL 60 Compilation, Academic Press, 1[illegible].

33. J. Myhill, "Linear Bounded Automata", WADD Technical Note 60-165.

34. D. H. R. Huxtable, On Writing an Optimizing Translator for ALGOL 60, in Introduction to System [illegible]gramming, ed. P. Wegner, Academic Press 1964.

35. IBM Laboratory Vienna, "Formal Definition of PL/I", TR 25.071, December 1966.

36. P. Wegner, The Structure of SNOBOL4, Technical Report No. 68-9, Cornell Computer Science Department, February 1968.