MAC TR-134

CSG MEMO-106

SEMANTICS OF DATA STRUCTURES AND REFERENCES

David J. Ellis

August 1974

This research was supported by the National Science Foundation under research grant GJ-34671.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

SEMANTICS OF DATA STRUCTURES AND REFERENCES

by

David J Ellis

Submitted to the Department of Electrical Engineering in September, 1974 in partial fulfillment of the requirements for the degrees of Master of Science and Electrical Engineer

ABSTRACT

Each programming language that handles data structures has its own set of rules for working with them. Notions such as assignment and construction of structured values appear in a huge number of different and complicated versions. This thesis presents a methodology which provides a common basis for describing ways in which programming languages deal with data structures and references to them. Specific concern is paid to issues of sharing.

The methodology presented here consists of two parts. The base language model, a formal semantic model introduced by Dennis, is used to give the work here a precise foundation. A series of "mini-languages" are defined to make it simpler and more convenient to express and describe the semantics for a variety of constructs found in contemporary programming languages.

THESIS SUPERVISOR: Jack B. Dennis

TITLE: Professor of Electrical Engineering

Acknowledgments

I wish to express thanks to my thesis supervisor, Professor Jack Dennis, for the many ways he helped me along in this work. He welcomed me into the Computation Structures Group when I was still looking for a group to join; brought the base language model to my attention; encouraged my ideas at every turn, even when I felt I was in a dead end; smoothed over numerous technical rough spots; and exhibited patience and acceptance throughout.

Thanks are due to Jack Aiello, Mark Laventhal and Nimal Amerasinghe, who read drafts of the thesis and made many helpful comments and suggestions. Anne Rubin provided technical assistance while I was typing the thesis.

Generous financial support was provided over the past three years by the Woodrow Wilson Fellowship Foundation, the M.I.T. Electrical Engineering Department, and Project MAC.

Finally, special gratitude goes to my family, for always giving me support and advice, standing by me in difficult times, and helping me overcome doubts about where I should be.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

1.1. General Goals

Students of computer science are confronted at a very early stage with a great variety of general-purpose programming languages. Descriptions of these languages place heavy emphasis on common features such as assignment, procedures, conditionals, input/output and block structure. Aside from variations in notation, there are numerous rules, exceptions and special cases which make for differences between comparable constructs in different languages. For example, the body of a DO-loop in FORTRAN must be executed at

FORTRAN	PL/1
N = 1 DO 50 I = 2,N . . [body] . . 50 CONTINUE	N = 1; DO I = 2 TO N; . . [body] . . END;
body executed once	body not executed

Fig. 1.1-1. Looping feature in two languages

least once, while in PL/1 it is to be skipped if the index is out of range (figure 1.1-1). Such differences can be studied by examining the semantics of different programming languages. The <u>semantics</u> of a programming language is the study of the meaning of its constructs, or in other words the effect of executing programs in the language. The particular concern of this thesis is the notion of <u>data structures</u> and the semantics pertaining to them as they appear in programming languages.

There are many areas of application in which the use of structured data is both helpful and convenient in problem solving. Some example areas are symbol manipulation, artificial intelligence, computer graphics, and simulation studies. Generally speaking, a data structure is an aggregate data object containing other data objects as components. Typical instances of data structures include arrays, sequences, vectors, tuples and lists. We will not dwell on the characteristics peculiar to each of these different varieties of data structure; our emphasis will be on more general properties relating to data structures and their components.

Typically, a programming language provides two basic operations for handling data structures: component objects of a data structure can be individually accessed and manipulated, and data structures can be constructed from designated objects as components. These operations interact with the assignment operation of a programming language in performing several other tasks, such as assigning structured values to identifiers, or updating components of a structure. There is a great similarity in appearance among constructs for performing such tasks in various programming languages. On the surface, from a casual examination of language descriptions, distinctions between analogous constructs in different languages appear to be mostly notational. But we shall see important semantic distinctions, particularly in the area of data being shared between different structures.

Since each programming language has its own set of rules for dealing with data structures and sharing, it is desirable to seek a rigorous method for describing what happens. Our goal, then, is to gain a more precise understanding of the semantics of data structures. This will provide a unified and coherent viewpoint for describing the different approaches to data structures as they are found in programming languages. We will pay specific attention to the difficult and important issue of properties of sharing. These issues depend ultimately on the concepts of cells (which model computer memory locations) and references to cells. References are also commonly known as pointers. We will first discuss general questions of programming language semantics, and then move towards a more specific treatment of data structures and references.

1.2. Background on Formal Semantics

A programming language provides a notation in which the programmer can model computational processes and the information on which they operate. Programming language semantics deals with the relationship between programs and the objects they represent. A formal semantics for a programming language is a precise description of such a relationship. There has been much study of formal semantics of programming languages. Wegner [Weg 72a] distinguishes three classes of formal semantic models:

(1) Abstract semantic models. In this approach, the objects being modeled are treated as mathematical entities independent of any particular representation. Models of

this class aim towards providing a formal mathematical description of the computational notions being studied. One well-known example of this approach to semantics has been the use of the lambda calculus as a semantic model for programming languages. The lambda calculus, which is described in [Der 74, Morr 68, Weg 68], is basically a mathematical formalism for the definition and application of functions. It is ideally suited for describing so-called <u>applicative</u> features of programming languages, such as evaluation of expressions, use of procedures, and block structuring. Landin demonstrated its usefulness in these areas [Lan 64] and presented a scheme for extending the lambda calculus formalism to model the language ALGOL 60 [Lan 65]. More recently, different extensions of the lambda calculus have been devised for describing data types [Reyn 73].

A second major example of the abstract approach to semantics is found in the work of Scott [Scot 70, Scot 71]. Scott makes use of the mathematical theory of lattices [San 73] to construct sets which are the domains of functions that represent the behavior of programs. The Scott formalism has been used recently to describe the semantics of ALGOL 60 [Mos 74].

We can briefly summarize abstract semantic models by saying that they characterize the action of programs as functions over various domains.

(2) Input-Output models. Models of this class use statements of mathematical logic as assertions about the state of a computer system at various points during the execution of programs on it. The semantics of a program is viewed as the relation between input assertions (the state of the system before execution) and output assertions (the state after the program is run). This approach to semantics, more frequently called the axiomatic approach was developed by Floyd [Floy 67] and Hoare [Hoar 69, Hoar 71]; there has been much further work on it. Axiomatic semantics is most useful in proving correctness of programs, i.e. establishing that the effect of executing a program fulfills mathematical conditions the program is supposed to satisfy.

(3) Operational models. This approach to semantics concerns itself specifically with modeling the changing states of a computer system performing computations. Such a task is usually accomplished by means of a state-transition system, in which a state of the model represents the information in the computer system at a given time. The effect

of a program on its input data is reflected in the sequence of transitions of the model. It is important to observe that given a state-transition system corresponding to some program, the sequence of states that models the execution of this program defines the action of an interpreter for the program. For this reason, the approach to formal semantics using operational models is called _interpretive semantics_.

We can describe the way in which an interpretive semantic model gives the semantics for a program written in some source language. A _translator_ transforms the program into an equivalent program in another language which we call an _abstract language_. Programs in an abstract language are acted upon by an interpreter; this action results in a sequence of state transitions of the model. The semantics of the original source-language program is given by such a sequence of transitions. One reason we make use of translators is that source programs are usually represented as character strings rather than as data objects suitable for processing by the interpreter.

Although the use of interpreters to implement programming languages was (and still is) commonplace, McCarthy [McC 62] was the first to use an interpreter to _define_ a language (LISP). The semantics of LISP is given formally by an interpreter written in LISP. Landin [Lan 64, Lan 66b] uses an interpreter called the SECD machine to define the lambda calculus, even though the lambda calculus is a mathematical formalism with a rigorous definition of its own. A more recent discussion of definitional interpreters is found in [Reyn 72].

Of these three approaches to formal semantics of programming languages, the interpretive approach is best suited for our goals of understanding the semantics of data structures and references. In order to properly explain the semantics of a program that handles data structures, we will need to know how the data structures are formed, their composition, the relationships between the structures and their components, sharing properties, and other items of information. The best way to get a handle on this kind of information is to consider the state of the system at various moments during the execution of the program. The interpretive approach is the only one which lends itself directly to working with states of the system. Both of the other approaches are better suited for proving assertions about programs and establishing their correctness; but these issues are outside our main concern here. A treatment of data structures from the viewpoint of axiomatic semantics may be found in [Lav 74]. We will work towards developing an interpretive model to be used as a semantic foundation for dealing with the important issues of data structures and references.

The most prominent interpretive model for semantics is the VDL model. VDL, the Vienna Definition Language, is a metalanguage for writing interpreters of programming languages. VDL interpreters have been written for languages such as ALGOL 60 [Lau 68], PL/1 [Walk 69, Luc 69], BASIC, and PDP-8 machine language [Lee 72]. An elementary introduction to VDL may be found in [Weg 72b]. Just as LISP works with lists, VDL works with tree-like data objects (which we call labeled trees). The basic operation of the VDL model is as follows: for each source language whose semantics we wish to describe, we define a translator and an interpreter. The translator transforms a source language program into an <u>abstract program</u>, which is a form of labeled tree suitable for manipulation by the interpreter (for each source language the corresponding abstract language will be some set of labeled trees; the structure of an abstract program varies from language to language). The interpreter, which consists of VDL code, accepts a labeled tree as input and interprets the effect of the program on its input data. For different languages, different interpreters are defined.

The fact that VDL uses treelike data objects reduces its desirability as a semantic model for our work on data structures. We will be studying data structures in which components may be shared between different objects; VDL's labeled trees do not directly admit sharing of any kind. Thus in order to model in VDL structures such as we will study, it would be necessary to go through the inconvenience of simulating the memory of a computer. Since the study of sharing is fundamental to our work, it is desirable to work with objects in which sharing is represented directly. We therefore prefer for our goals a semantic model that manipulates data objects of a more general nature than VDL's labeled trees.

In [Denn 71], Dennis outlines an interpretive semantic model called the <u>base language model</u>. The data objects manipulated by this model are variants of directed graphs and can directly model sharing. As with VDL, for each language whose semantics we wish to describe, we must specify a translator which transforms programs in the language into data objects suitable for consumption by the model. These objects are called procedure structures in the base language model. Procedure structures, like VDL's abstract programs, are acted upon by the interpreter to produce state transitions. But the base language model differs from VDL in that the composition of a procedure structure generated by the translator from some source program does not depend on the language in which the program was written. As a result, there is no need to define a separate interpreter for each programming language. There is a single, pre-supplied interpreter for the base language model which accepts arbitrary procedure structures and interprets them as programs. Thus we see that the translators for the base language model translate programs from their respective source languages into a single, common language. We call this language the base language. A procedure structure represents a program in the base language, which consists of a sequence of instructions. The individual base language instructions specify the fundamental state transitions of the model.

In order to achieve the language-independence of the interpreter in the base language model, the translators must do more work than their VDL counterparts. A VDL translator simply converts a program from character string to labeled tree, while a translator for the base language model must perform functions similar to those of a compiler. Thus, once we specify the semantics of the base language, i.e. decide on a formal specification of the actions performed by the interpreter in the base language model, the semantics of a particular programming language is determined by its translation into the base language.

The base language model is extremely well suited for our work. The primitive instructions of the base language are particularly convenient for manipulating structured objects and dealing with sharing. We can view the base language as the machine language for a computer with heap-structured memory and symbolic address space. In this respect, programs in the base language will be similar to conventional assembly language programs. This similarity is a source of further convenience in using the base language as a programming tool.

Amerasinghe [Amer 72] described the translation of a block-structured language BLKSTRUC into the base language. In BLKSTRUC, procedures are "first-class objects" [Stra 67] which can be used in contexts as general as objects of other types. BLKSTRUC's treatment of procedures is more general than ALGOL 60's. The action of a translator for a language with non-local goto's is described in [Amer 73]. Translators for the languages SNOBOL4 and Simula 67 are discussed in [Dra 73] and [Cou 73]. These works show the use of the base language model in describing the semantics of various powerful programming languages. We will be using a version of the base language model as the semantic foundation for our study of data structures.

1.3. Plan for the Thesis

We outline here the topics covered in the rest of this thesis. Chapter 2 describes the base language model as we will be using it. The action of the interpreter is given by describing the effect of the instructions of the base language. The approach in Chapter 2 is informal; a more rigorous treatment is found in the Appendix. Once the behavior of the base language interpreter is known, we have a handle on the semantics of the programming-language constructs that interest us. All that will then need to be done to supply a formal semantic definition is simply to describe the action of a translator which produces base language code.

In the remainder of this thesis we will be using the base language model as a semantic foundation for describing the different ways various programming languages deal with data structures. We want to make clear distinctions between comparable constructs in different languages. Although the semantics of data structuring constructs can be precisely expressed by using the base language model, there is a certain respect in which the model is less than ideal as a descriptive vehicle. Data structures as they are found in programming languages are tied up with the notions of variables and values. We would like to make use of these notions in talking about the semantics of data structures. But the descriptive level of the base language is only equipped for talking about primitive transformations on the objects which comprise the interpreter states. In this sense the base language is too "low-level" for describing data structures in a manner suitable for our purposes.

To provide a better descriptive mechanism, we will follow the approach taken by Ledgard [Led 71] in defining a series of "mini-languages." Mini-languages provide descriptive levels appropriate to our needs, yet at the same time avoid the syntactic and semantic complexity of full-scale programming languages. The primary advantage of the mini-language approach is that we can isolate the concepts we wish to describe by eliminating all the conceptually extraneous notions that are needed in a full-size language. Accordingly, in a mini-language for describing data structures, there are no procedures, conditional expressions, loops, goto's or operators. Mini-languages are not meant to be viable languages for actual programming; they are used for descriptive purposes only. The syntax and semantics of a mini-language are simple enough to be readily understood on an informal basis; the semantics can then be formalized by specifying translation into the base language. In this manner, the semantics of data-structuring constructs in full-scale programming languages can be given by describing how to express these notions in a suitable mini-language.

Chapter 3 presents mini-languages for describing the notions related to assignment, data structures, pointers and sharing. These mini-languages are then used to describe the data structuring semantics of several full-scale programming languages.

In Chapter 4, we treat the additional notion of static typechecking, which has a direct bearing on the semantics of data structures in many important programming languages. This notion of static typechecking differs from Ledgard's in that it deals with structured types, where Ledgard [Led 71] deals with functional types and the types of arguments and returned values. As in Chapter 3, we treat the data structuring facilities of three full-size languages; in these languages the concept of static typechecking is directly tied in with the semantics of data structures (specifically assignment).

Chapter 5 presents a summary of what we cover in this thesis and suggests extensions for further study.

Chapter 2

THE BASE LANGUAGE MODEL

2.1. Overview of the Model

We have chosen as the semantic foundation for our work a version of the base language model set forward in [Denn 71] and [Amer 72]. The base language model centers around a base language interpreter, which is essentially a state-transition system that we shall use to express the meaning of computations. The interpreter specifies the behavior of an entire computer system. We represent a computation by a sequence of interpreter states. A state of the interpreter will be a certain kind of mathematical object embodying the information contained in the computer system at a particular point in time. We shall define a base language called BL each of whose programs consists of a sequence of instructions. Each instruction specifies a functional transformation between interpreter states. The language BL is adapted from the rudimentary language described by Dennis in [Denn 71].

We represent interpreter states by mathematical objects known as BL-graphs. Suppose we are given a set ELEM of elementary objects and a set SEL of selectors. (For our purposes, ELEM consists of integers, real numbers and strings; SEL consists of integers and strings.) Then a BL-graph is a variant form of directed graph; it consists of nodes and arcs. Each arc connects two nodes in a specified direction and is labeled with a selector. We may associate an elementary object with each node from which no arcs lead out. There must also be a distinguished subset of the nodes (called the root nodes) from which each node of the graph can be reached along some directed path of arcs. We give a formal mathematical definition of BL-graphs in the Appendix.

A BL-graph with a single root node is called a BL-object. We identify a BL-object by its root node. Specifically, for any node α in a BL-graph G, we associate with α the subgraph of G whose nodes and arcs are accessible from α. This subgraph is a BL-graph with α as its root node; we call it the object of α.

If there is a directed path from one node of a BL-graph to another node, then the second node is called a descendant of the first node. All nodes in a BL-graph are descendants of some root node. A node from which no arcs emerge is called a _leaf node_. An elementary object attached to a leaf node is called the _value_ of that node. If there is an arc from a node α to another node β, then β is called a _component_ of α, and the object of β is called a component of the object of α. Components are named by the selectors on the arcs leading into them. If an object is a component of two distinct objects, it is said to be _shared_ between them. Nodes in a BL-object are denoted by _pathnames_. A pathname for a node is a sequence of selectors labeling a directed path to that node from the root node. If the object of a node is shared, then the node will have distinct pathnames. The property of sharing is of major significance; we will have much to say about it.

We will be making heavy use of pictorial representations of BL-objects. An elementary object is drawn as an encircled value (figure 2.1-1).

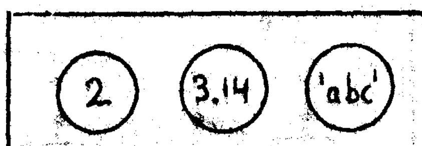


Fig. 2.1-1. Sample elementary objects

For a general BL-object, the nodes are drawn as heavy dots. The root node is at the top. Arcs emerging from a node are drawn downwards from a horizontal line attached to the node. Selectors are written across the arcs that they label. If a selector is a string, we do not enclose it in quotes. Elementary objects attached to root nodes hang downwards from them. Thus our pictorial conventions for BL-objects differ slightly from those used in [Denn 71].

Sample BL-objects are pictured in figures 2.1-2 and 2.1-3. The object in figure 2.1-2 has three components, named k, c and a. The c-component is empty. The k-component has two components, both of which are leaf nodes. The leaf node with value 9 has pathname k.c. The leaf node with value 'hi' is shared between nodes k and a and has pathnames k.u and a.6. In figure 2.1-3, the object with value 1.6 is shared between the objects s.b and s and has pathnames s.b.5 and s.4. The object of node c is shared

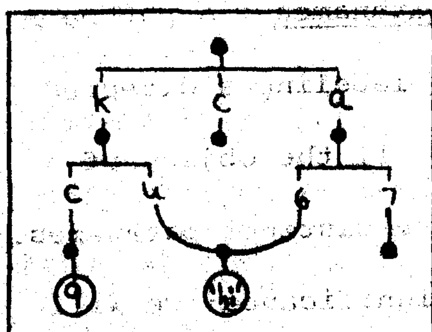


Fig. 2.1-2. A sample BL-object

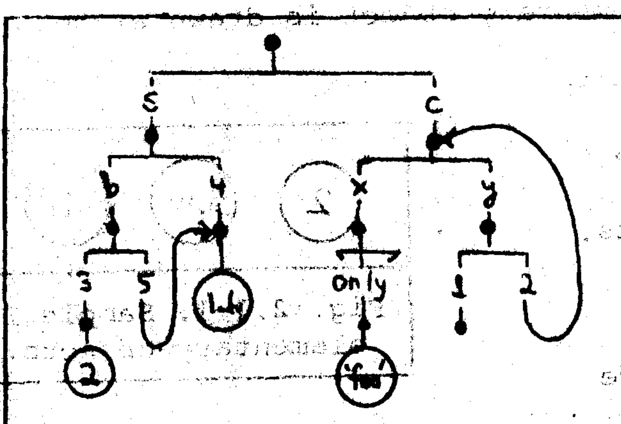


Fig. 2.1-3. A sample BL-object

between the object of the root node and the object c.y. Since the node c is a descendant of itself, it has infinitely many pathnames c, c.y.2, c.y.2.y.2, c.y.2.y.2.y.2, and so on. The path joining this node to itself is a directed cycle.

A basic difference between out BL-graphs and the graphs of [Denn 71] is that Dennis does not allow directed cycles in his objects. Cycles seem to impair the management of storage and the handling of parallelism in computation. However, cycles occur in many of the structures we shall be modeling. Moreover, they are difficult to detect and remove (see [Amer 72] for more details on the problems of cycles). We shall therefore not rule out cycles here.

We follow [Denn 71] in giving the structure of a BL-object which represents a state of the interpreter. An interpreter state is a BL-object having three components as follows:

(1) The universe-component models system-resident information, both data and procedures. Generally speaking, this information is independent of which computations are currently active or how far various computations have progressed.

(2) The local-structure-component of an interpreter state has as components a series of activation records for the various procedures being interpreted in the system. These components are called local structures; there is one local structure for each activation of each base language procedure. A local structure represents the environment for its activation, primarily identifiers and their associated values. Thus the local-structure component of an interpreter state records the progress of computations by modeling their changing environments.

(3) The control-component has as components a number of sites of activity, which indicate for each current computation the next instruction to be executed, the appropriate environment (local structure) for the computation, and other information.

We shall not go into the details here of representing the universe- and control- components of interpreter states. The interested reader can consult the Appendix for that kind information. We will be dealing almost exclusively with local structures in the remainder of this chapter. In the next section, we describe the action of a number of primitive BL instructions.

2.2. Base Language Instructions

We introduce the primitive instructions of BL, which define state transitions of the interpreter in our model. Each BL instruction executed by the interpreter belongs to some procedure written in BL and is interpreted during an activation of the procedure. We call the local structure corresponding to this activation the current local structure (c.l.s.) for the instruction.

A BL instruction consists of an operation code and up to three operands. The operation code is underlined. Most of the operands of the various instructions are selectors, which are frequently used to denote names of components of the root node of the c.l.s. We reserve the letters x, y, and z for selector names used in this fashion.

We shall give informal descriptions of the effects of BL instructions, accompanied by sample "before" and "after" diagrams of the c.l.s. A more formal definition of these instructions may be found in the Appendix.

Each instruction is designed to perform a specific function in changing the c.l.s. This is called the primary role (or, more simply, the role) of the instruction, and depends on certain conditions being fulfilled (e.g. the presence or absence of specific components in the c.l.s.). The effect of an instruction when such conditions do not hold is called a subsidiary effect, or subeffect.

The create instruction is used to create a new component in the c.l.s. Provided that the c.l.s. has no x-component, the primary role of the instruction create x is to add one (figure 2.2-1). The new x-component will be an empty leaf node. If the c.l.s. already has an x-component, then the instruction create x has a subsidiary effect of changing the arc with selector x from the root node to point to a newly allocated node. For this subeffect the former x-component node will remain as part of the c.l.s. only if it was shared with some other node. Figures 2.2-2 through 2.2-4 illustrate subeffects of the instruction create x and its interplay with the sharing property. Portions of a diagram enclosed in dotted lines are no longer part of the c.l.s.

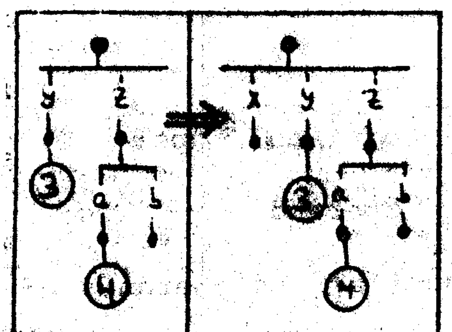


Fig. 2.2-1. role of create x

and can be thought of as garbage-collected.

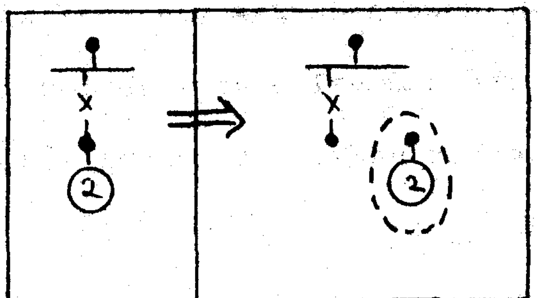


Fig. 2.2-2. A subeffect of <u>create</u> x

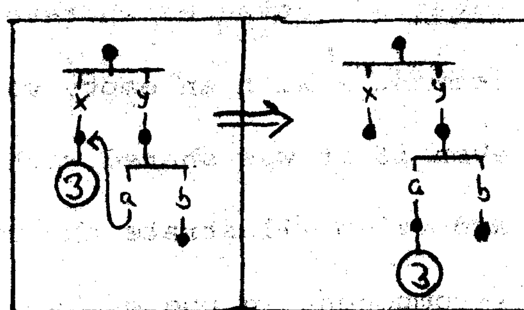


Fig. 2.2-3. A subeffect of <u>create</u> x

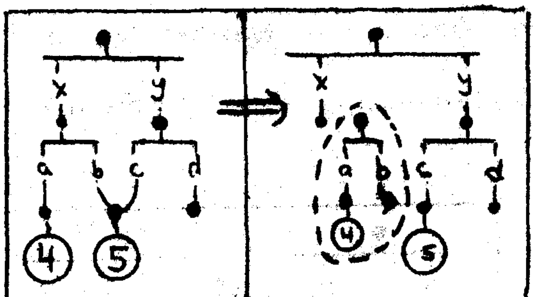


Fig. 2.2-4. A subeffect of <u>create</u> x

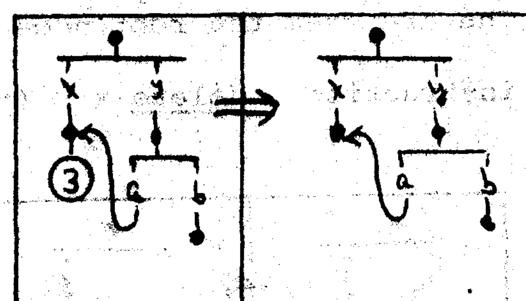


Fig. 2.2-5. Role of <u>clear</u> x

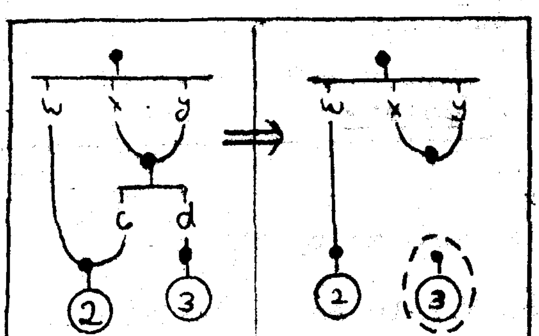


Fig. 2.2-6. Role of <u>clear</u> x

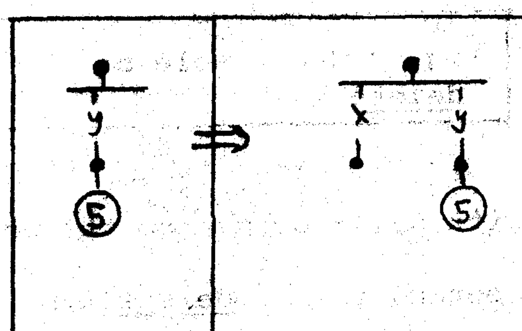


Fig. 2.2-7. A subeffect of <u>clear</u> x

The clear instruction is used to make a node empty; clear x detaches whatever hangs downward from the node x, leaving x with an empty value. The old value of x is lost even if it was shared with some other node. Figures 2.2-5 and 2.2-6 illustrate the role of clear x. If there is no x-component in the c.l.s., clear x acts like create x and generates one (fig. 2.2-7).

The delete instruction removes arcs from the c.l.s. The arc from the root node to the node x is removed by the instruction delete x (figs. 2.2-8 and 2.2-9). The arc

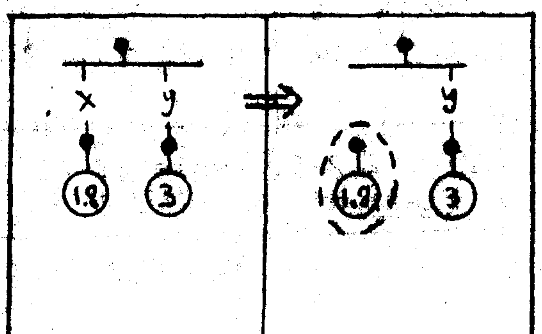


Fig. 2.2-8. Role of delete x

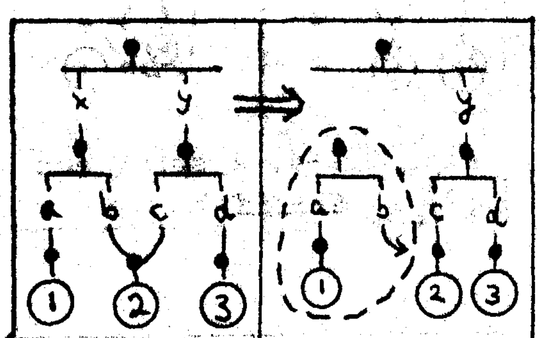


Fig. 2.2-9. Role of delete x

with selector m from the node x is removed by the two-operand form delete x,m (figs. 2.2-10 and 2.2-11). If an arc to be removed does not exist, then the subeffect of the delete instruction is that no action be taken.

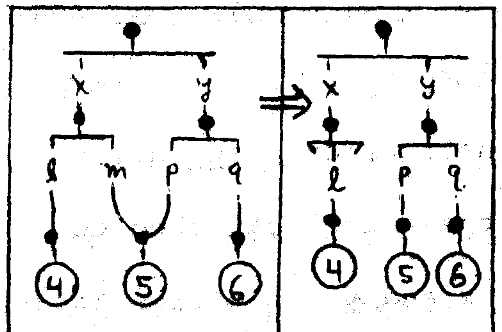


Fig. 2.2-10. Role of <u>delete</u> x,m

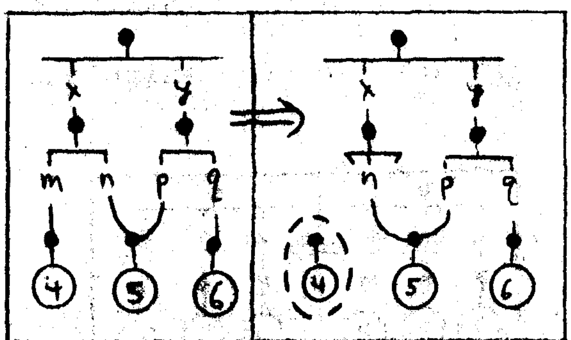


Fig. 2.2-11. Role of <u>delete</u> x,m

The <u>const</u> instruction is used to attach elementary objects to nodes. If v is any elementary object, then <u>const</u> v,x causes the value v to be attached to the node x. The old value of x, if any, is lost. Figure 2.2-12 illustrates the role of the instruction <u>const</u> 5,x (where x is a leaf node), and figure 2.2-13 shows a subeffect of the same instruction (for the case when x is not a leaf node).

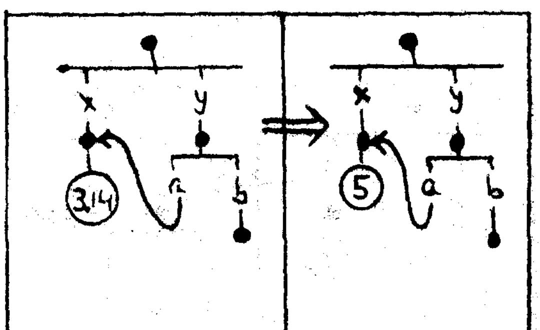


Fig. 2.2-12. Role of <u>const</u> 5,x

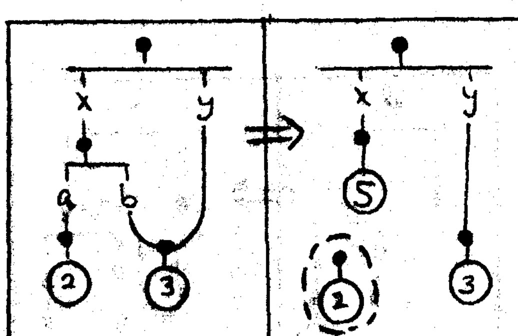


Fig. 2.2-13. Subeffect of <u>const</u> 5,x

Arithmetic instructions such as *add*, *subtr*, *mult* and *div* are used to manipulate elementary values. For example, the instruction *add* x,y,z adds the values attached to nodes x and y and places the sum in node z (figure 2.2-14). It is an error to attempt to execute an arithmetic instruction if one of the first two operand nodes fails to exist or contains an improper value (not a leaf node or empty or wrong type of elementary object). We leave the effect of such an attempt undefined.

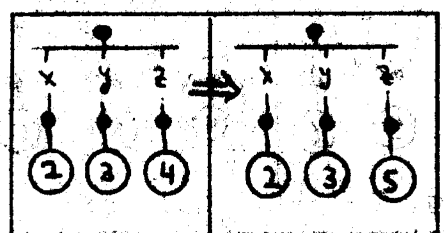


Fig. 2.2-14. Role of *add* x,y,z

The *link* instruction is used to initiate sharing between nodes. The instruction *link* x,n,y causes the node y to become the n-component of x (so that y will be shared

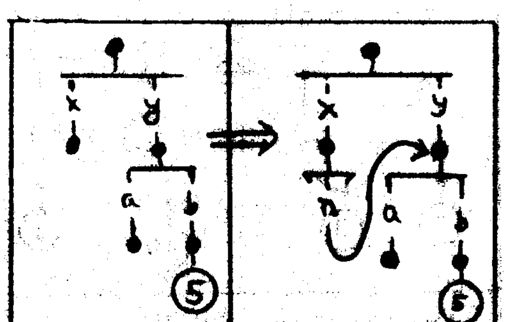


Fig. 2.2-15. Role of *link* x,n,y

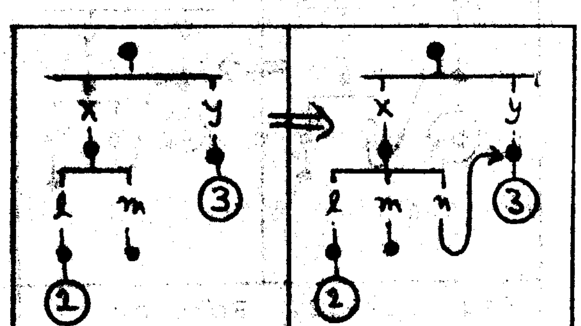


Fig. 2.2-16. Role of *link* x,n,y

between the node x and the root node). This is done by adding an arc with selector n from node x to node y. Figures 2.2-15 and 2.2-16 illustrate the role of the instruction <u>link</u> x,n,y. If x already has an x-component or is a leaf node with some elementary value, then the subeffect of the same instruction causes the old value of x to be lost (figs. 2.2-17 and 2.2-18). The nodes for x and y must be present or else the instruction is illegal.

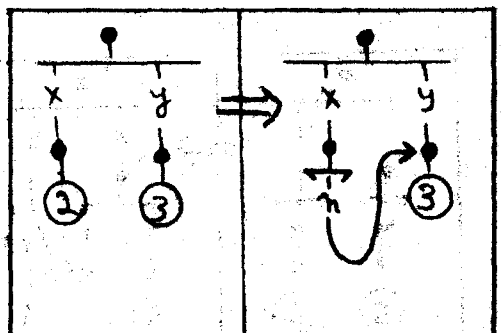


Fig. 2.2-17. Subeffect of <u>link</u> x,n,y

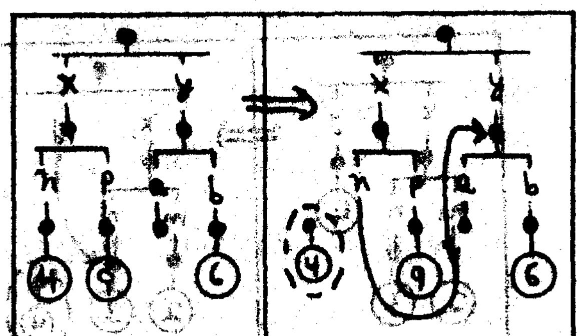


Fig. 2.2-18. Subeffect of <u>link</u> x,n,y

The <u>select</u> instruction satisfies a dual purpose. If a node x has an n-component, then the instruction <u>select</u> x,n,y makes the n-component of x the y-component of the root node (so that it can now be "addressed" by further BL instructions). In this manner a BL procedure may gain access to arbitrary nodes of a c.l.s. If x has no n-component, then the instruction <u>select</u> x,n,y generates one first, then makes it the y-component of the root node. This is the principal way to construct BL-objects, i.e. by using the <u>select</u> instruction to add on components. These two roles of the <u>select</u> instruction are depicted in figures 2.2-19 and 2.2-20, respectively. The root node may or may not have a y-component prior to the execution of <u>select</u> x,n,y. If it does, then the value is lost unless it was shared.

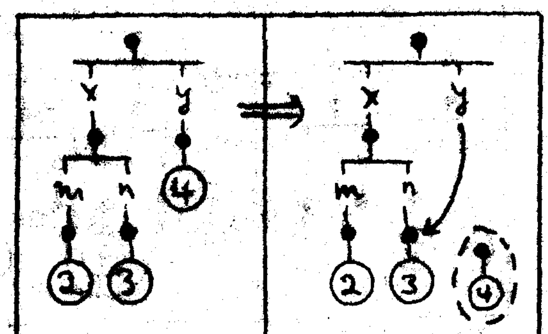


Fig. 2.2-19. 1st role of <u>select</u> x,n,y

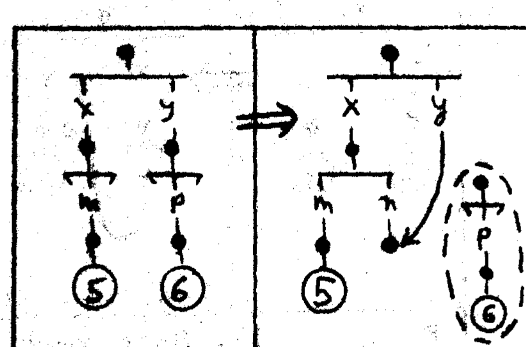


Fig. 2.2-20. 2nd role of <u>select</u> x,n,y

The <u>apply</u> instruction provides for the activation of BL procedures. Let the p-component of the c.l.s. represent the BL code for some procedure (i.e. be a procedure structure). Then the instruction <u>apply</u> p,x activates this procedure in the following manner: First, a new, empty local structure is created. The x-component of the c.l.s. is then made the $par-component (parameter linkage) for the new local structure (we refer to the BL-object x as an <u>argument structure</u>). Finally, control is passed to a new site of activity. This means that the newly-created local structure becomes the c.l.s. and the old site of activity is made dormant. The interpreter will now execute instructions from the procedure p until it is told to return.

The <u>return</u> instruction provides for termination of the execution of a BL procedure and for return to the calling procedure. Upon execution of a <u>return</u> instruction, the c.l.s. is deleted. All its components vanish. The parameter linkage, since it shares with the argument structure of the invoking procedure's local structure, remains. Control is returned to the dormant site of activity for the invoking procedure, and its local structure becomes the new c.l.s. The invoking procedure resumes from where it left off.

In order to invoke a procedure, it must be represented as a component of the c.l.s. The <u>move</u> instruction makes data in the universe available for invocation as a BL procedure. We will not have occasion to use this instruction here; further details are found in the Appendix.

The instructions of a BL procedure are labeled with natural numbers; execution of a BL procedure consists of the successive execution of its instructions in sequence according to the numbers labeling them. The remaining BL instructions provide for changes in the control sequence. Each of them has as one of its operands a label ℓ which must be a natural number labeling some instruction of the procedure currently being executed.

The instruction <u>goto</u> ℓ transfers control to the instruction in the current procedure whose label is the natural number ℓ.

The instruction <u>elem?</u> x,ℓ tests whether the x-component in the c.l.s. is a leaf node (<u>elem</u>entary object). If <u>not</u>, control passes to instruction number ℓ.

The instruction <u>empty?</u> x,ℓ checks whether the x-component of the c.l.s. is an empty leaf node (i.e. no components and no elementary value). If <u>not</u> empty, control transfers to instruction number ℓ.

The instruction <u>nonempty?</u> x,ℓ performs the same test as the corresponding <u>empty?</u> instruction, but control passes to ℓ if the x-component is empty.

The instruction <u>eq?</u> x,y,ℓ looks at the x- and y-components of the c.l.s. Both must be leaf nodes, or else the effect of this instruction is undefined. These nodes are checked to see if they have the same elementary value. If the test <u>fails</u> (i.e. their values are <u>not</u> equal), then control passes to ℓ.

The instruction <u>has?</u> x,m,ℓ checks whether the x-component object of the c.l.s. has an m-component. If <u>not</u>, control passes to ℓ.

The instruction <u>same?</u> x,y,ℓ checks whether the x- and y-components of the c.l.s. share the same node. If <u>not</u>, i.e. they are distinct nodes, control passes to ℓ.

In all the above conditional instructions, if the c.l.s. fails to have a component indicated by some operand, then the effect is undefined.

Other conditional instructions analogous to the above ones can be defined (e.g. testing whether one elementary value is less than another). We will have no need here for such additional instructions.

Finally, we discuss one more instruction that will be needed. Given a BL object, we will want to be able to access each of its components, without knowing beforehand the names of the selectors. The _getc_ instruction serves this purpose. Successive executions of the same instruction _getc_ x,i,ℓ extract successive components of the x-component of the c.l.s. by causing the i-component of the c.l.s. to assume as its successive values the selectors on the arcs leading from the node x. No component will be extracted more than once, and control passes to ℓ when no more components of x remain to be accessed.

2.3. Programming Conventions for BL

In this section we introduce a few programming conventions which will make BL procedures easier to write and understand. We can view BL as the machine language for a hypothetical computer. Our conventions are then similar to the programming features provided by a macro-assembler.

Although individual instructions in a BL procedure are labeled by natural numbers, we shall use symbolic labels. For example, suppose that x and y denote leaf nodes in the c.l.s. Then the BL code of figure 2.3-1 places the

```
        eq?     x,y,no
        const   'yes',ans
        goto    skip
no:     const   'no',ans
skip:   ....
```

Fig. 2.3-1. Use of symbolic labels in BL string value "yes" in the node ans if the values of x and y are equal, "no" if they aren't.

The nodes addressed by operands in the BL instructions must be direct components of the root node of the c.l.s. With the <u>select</u> instruction, we can access nodes further down in the c.l.s. For instance, suppose we wish to change the value 3 in figure 2.3-2 into the value 4. This is done by the <u>const</u> instruction, but in order to access the proper node, we must use the <u>select</u> instruction three times. In the BL code that performs our task (figure 2.3-3), the reserved selector $temp acts as a temporary variable. By using a "dotted pathname" convention to refer to appropriate nodes, we can abbreviate this BL code as the single instruction <u>const</u> 4,x.b.d.e. This can be viewed as a macro-instruction whose expansion gives the required <u>select</u> instructions. Alternatively, we can look at

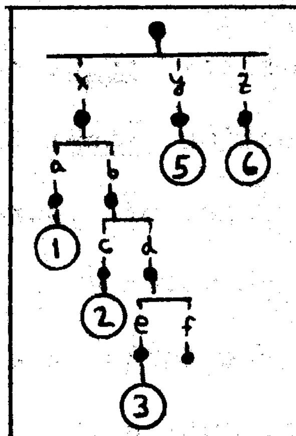


Fig. 2.3-2.

```
select   x,b,$temp
select   $temp,d,$temp
select   $temp,e,$temp
const    4,$temp
```

Fig. 2.3-3. BL code to access a node

this convention as extending "addressability" to arbitrary nodes in the c.l.s.

We will make frequent use of a macro-substitution capability, which is provided by a "*" convention. If z is a leaf node containing some elementary value, then *z denotes this elementary value. For example, in the c.l.s. of figure 2.3-2, *z denotes the value 6. The abbreviation const *z,y specifies the same transition as the instruction const 6,y when the c.l.s. is in this state. In the c.l.s. of figure 2.3-4, the leaf node with value 2 can be addressed by any of the forms x.a, x.*z, *y.a, or *y.*z, while the value 2 itself can be denoted by any of the forms *(x.a), *(x.*z), *(*y.a), or *(*y.*z). As a third example, the BL code of figure 2.3-5 sets all the components of the object x to zero. Note that the leaf node i contains as successive values the names of the selectors from x. Thus the dotted pathname x.*i refers to the successive com-

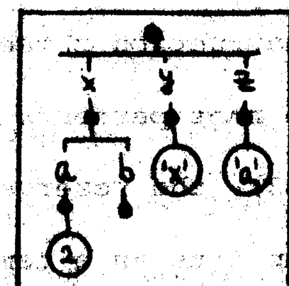


Fig. 2.3-4.

```
loop:   getc    x,i,out
        const   0,x.*i
        goto    loop
out:    ....
```

Fig. 2.3-5.

ponent nodes of x.

We now define several macros for BL to denote commonly performed functions. The .setl macro (set up local structure) is used to set up new components in the c.l.s. Figure 2.3-6 shows the definition of the .setl macro, and figure 2.3-7 gives an example of its effect.

```
.setl   (x1,...,xn)

create  x1
      :
create  xn
```

Fig. 2.3-6. Expansion of .setl macro

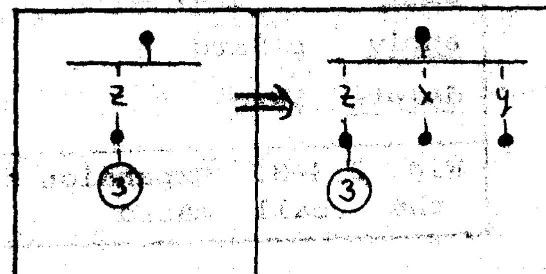


Fig. 2.3-7. Effect of .setl (x,y)

The remaining macros we will use deal with linkage between BL procedures. We first define a procedure closure to be a BL-object with two components. The $text-component contains BL text of a procedure, and the $env-component contains references to the global variables named in the procedure. (Note that "$" is a legal character in BL.)

The .call macro expands into BL code to invoke a procedure. In the definition in figure 2.3-8, the node p must be a procedure closure, and a1, ... , an are selectors leading to the arguments, which may be arbitrary BL-objects.

```
.call    p,(a1,...,an)

create   $arg
link     $arg,$glob,p.$env
link     $arg,1,a1
   .
   .
link     $arg,n,an
apply    p,$arg
delete   $arg
```

Fig. 2.3-8. Expansion of the .call macro

Figure 2.3-9 gives an example of the invocation of a procedure p having a single global reference w; the procedure p is called with arguments x and y. The "old c.l.s." is the local structure of the invoking procedure, and the "new c.l.s." is the local structure of the called procedure p. The "after" picture shows both the old c.l.s. and the new c.l.s. when control is passed to the procedure p.

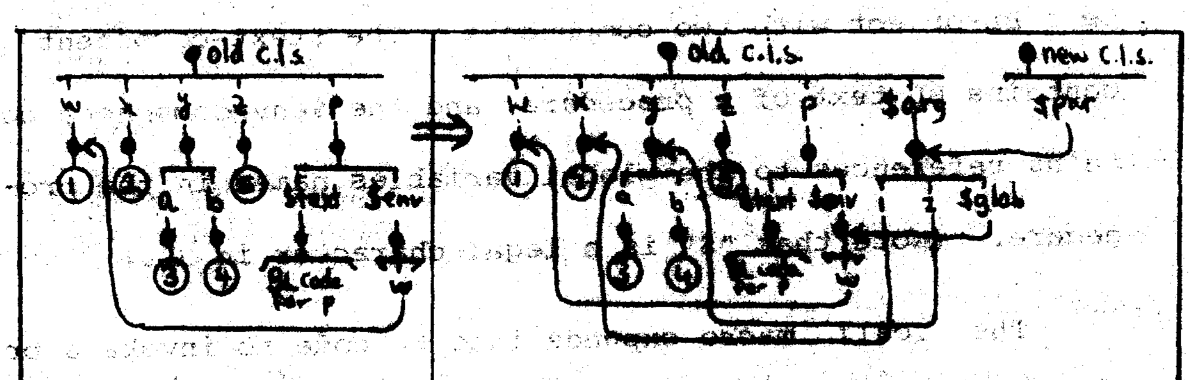


Fig. 2.3-9. Effect of .call p,(x,y)

The .getp macro (get parameters) serves to bind the formal parameters of a procedure to the actual arguments with which it was invoked. The .getg macro (get globals) makes the global variables named in a procedure accessible in its body. These two macros are defined in figures 2.3-10 and 2.3-11.

```
.getp   (x1,...,xn)

select   $par,1,x1
    :
select   $par,n,xn
```

Fig. 2.3-10. Expansion of the .getp macro

```
.getg   (x1,...,xn)

select  $par.$glob,x1,x1
    :
select  $par.$glob,xn,xn
```

Fig. 2.3-11. Expansion of the .getg macro

The first actions a procedure normally performs when given control are the retrieval of parameters and global variables (using the .getp and .getg macros respectively). Figure 2.3-12 is a "continuation" of figure 2.3-9, showing both c.l.s.'s after the invoked procedure p executes the two macros .getp (u,v) and .getg (w).

With the BL programming conventions that have been defined here, we are now ready to use BL as the language of our semantic model.

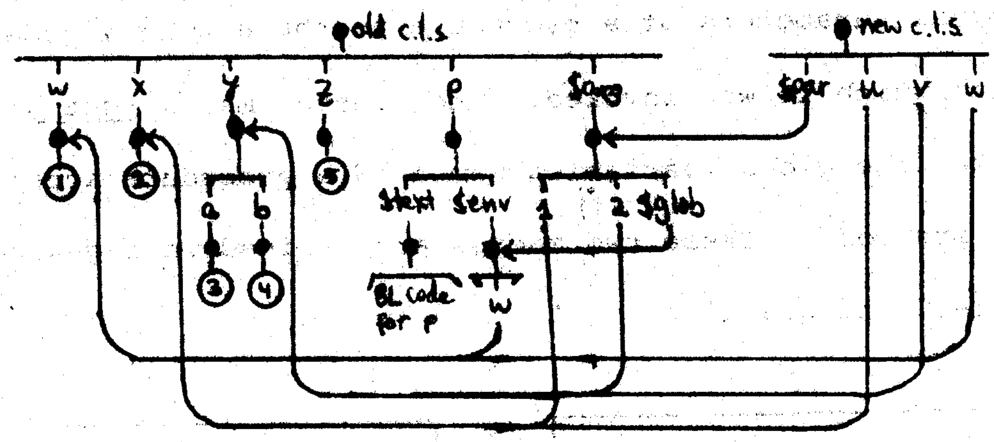


Fig. 2.3-12. State of the two c.l.s.'s after procedure p executes the macros .getp (u,v) and .getg (w)

Chapter 3

STRUCTURES, POINTERS AND SHARING

3.1. Mini-Languages

In this chapter we present a series of mini-languages which treat the issues of structures, pointers and sharing. The progression of mini-languages is hierarchical in that it starts from a few basic concepts and proceeds outward by extension. Mini-Language 0 is the "kernel" language, isolating the notions of variables, values and assignment. These basic concepts form the core for our domain of discourse. Mini-Language 1 is a direct extension of Mini-Language 0, adding to it structured values and the notions of construction of structured objects and selection of components from structures. Mini-Language 2 extends Mini-Language 1 by including pointers and the two operations of building and following pointers. Finally, Mini-Language 3 treats the idea of sharing of components between objects. By revising the concept of structured value found in Mini-Language 1, the notions relating to pointers are subsumed in Mini-Language 3 by notions relating to sharing.

Each mini-language is treated in a separate section of this chapter. In each section, we first discuss in general terms the concepts addressed by the mini-language under consideration. New terminology is introduced, and we describe the relation to previous and/or succeeding mini-languages. We then supply a BNF-style syntax together with a description of the syntactic classes and what they represent. The semantics of the mini-language is stated informally, a la ALGOL 60. We then formalize the semantics by giving samples of rules for translation from the mini-language into the base language BL. Each section is concluded by a "movie" illustrating the interpretation of the BL program produced by the translator from a sample program in the mini-language.

The final section of this chapter applies these mini-languages to the task of describing the data structuring semantics of "real-world" programming languages. The languages PAL, QUEST and SNOBOL4 are used as examples.

3.2. Mini-Language 0 -- Basics

Mini-Language 0 (ML-0) is the foundation upon which we build our mini-language setup. In introducing the concepts of value, location and assignment, ML-0 serves as a kernel for our set of mini-languages. The notions of structures, pointers and sharing will emerge as extensions to ML-0 in succeeding mini-languages.

All our mini-languages, starting with ML-0, operate within the conceptual world of values stored in locations which we call cells. The relationship between a cell and the value stored in it is called the contents mapping. A cell with no value stored in it is said to be empty and has no contents. We are concerned here with the fundamental operation of assignment, which is used to change the contents mapping. In fact, the entire purpose in creating ML-0 was to isolate the concept of assignment by placing it in as minimal and austere a set of surroundings as possible. This notion of assignment will remain unchanged in the remaining mini-languages of this chapter. The assignment statements of these languages will be "consistent" extensions of what we define in this section.

Another important concept we deal with here is the notion of binding. Each identifier in an ML-0 program is associated with a unique and distinct cell. This association is called the binding of an identifier. The value of an identifier will be the contents of the cell to which it is bound. (An identifier bound to an empty cell has no value.) Unlike the contents mapping, the binding relation remains invariant throughout the execution of an ML-0 program. This invariance is a property not only of ML-0, but of all the mini-languages in this thesis.

Syntax of ML-0

We give a BNF-style syntax for ML-0. Informal use is made of the ellipsis ("...") to indicate repetition. Two syntactic classes are primitive: ⟨integer⟩ denotes integer constants, and ⟨identifier⟩ denotes alphanumeric strings starting with a letter.

```
⟨program⟩     ::=  ⟨assignment⟩ ; ... ; ⟨assignment⟩
⟨assignment⟩  ::=  ⟨destination⟩ ← ⟨expression⟩
⟨expression⟩  ::=  ⟨destination⟩ | ⟨generator⟩ | nil
⟨destination⟩ ::=  ⟨identifier⟩
⟨generator⟩   ::=  ⟨integer⟩
```

Description

To understand assignment, we explain the syntactic classes relating to values and cells. A ⟨generator⟩ is a piece of program text denoting a value. All values in ML-0 are integers; subsequent mini-languages include other types of values as well. A ⟨destination⟩ is a piece of program text referring to a cell; ⟨destination⟩s in ML-0 are simply

⟨identifier⟩s, i.e. variable names. The reserved word nil will be used to signify empty cells. An ⟨expression⟩ is a piece of program text which "yields" a value. The semantic description below discusses evaluation of ⟨expression⟩s in ML-0.

An ML-0 ⟨program⟩ is simply a sequence of ⟨assignment⟩s, each of which consists of a ⟨destination⟩ and an ⟨expression⟩. The basic meaning of an ⟨assignment⟩ is to cause the value yielded by the ⟨expression⟩ to be stored into the cell referred to by the ⟨destination⟩.

Semantics of ML-0 (informal)

The notions we have just introduced will now be made more precise. We give the semantics associated with each significant syntactic class of ML-0 (now as a description in English, later more formally via translation into BL).

(1) ⟨program⟩s: The execution of an ML-0 ⟨program⟩ consists of two steps. First bind each ⟨identifier⟩ occurring in the ⟨program⟩ to a distinct, empty cell. Then execute all of the ⟨assignment⟩s sequentially, left to right. This rule giving semantics of ⟨program⟩s will remain intact for all the subsequent mini-languages in this chapter.

(2) <u><assignment>s</u>: The execution of an <assignment> consists of three steps --

(i) Identify the cell referred to by the <destination> on the left-hand side of the <assignment> (see rule (3) below).

(ii) Obtain the value yielded by the <expression> on the right-hand side (see rule (4) below).

(iii) Make the value from step (ii) the new contents of the cell from step (i).

Thus the effect of executing an <assignment> is a change in the contents mapping. This rule, like rule (1), will govern the semantics of the remaining mini-languages.

(3) <u><destination>s and <identifier>s</u>: A <destination> in ML-0 is always some <identifier>, and refers to the cell bound to this <identifier>. This binding is determined at the beginning of program execution; as we have already said, it remains constant throughout execution.

(4) <u><expression>s</u>: There are three varieties of <expression> in ML-0. We describe their semantics in rules (5), (6) and (7) below.

(5) <u>nil</u>: The special symbol <u>nil</u> indicates the absence of a value. Any time we are directed to store in some cell the value yielded by an <expression> which is <u>nil</u>, this means to make the cell empty. All of our mini-languages

treat nil in precisely this manner.

(6) ⟨destination⟩s as ⟨expression⟩s: When a ⟨destination⟩ occurs as an instance of an ⟨expression⟩ (in ML-0, this means on the right-hand side of an ⟨assignment⟩), it yields the value contained in the cell to which it refers (see rule (3) above). If this cell is empty, the ⟨expression⟩ is treated like nil (see rule (5) above). This semantic rule (known elsewhere as "dereferencing") will hold verbatim for all our mini-languages.

(7) ⟨generator⟩s: A ⟨generator⟩ in ML-0 is an ⟨integer⟩, which is the decimal representation of some integer value. It is this value which is yielded by the ⟨generator⟩.

The above seven rules constitute our informal description of the semantics of ML-0.

BL Representation

The semantic rules we just gave are a bit long-winded and imprecise. A rigorous description of the semantics of ML-0 can be obtained by "translating" these rules into BL instruction sequences. Before doing this, we discuss our basic conventions for representing mini-language programs in the base language model. To each program in one of our mini-languages, there is a single local structure. The cells used by the program are represented by nodes in the local structure. For each identifier occurring in the program, there is a correspondingly named component of the local structure which gives its binding. In other words, the cell bound to an identifier x will be the x-component node of the local structure. The contents of this cell is the object of its node. Thus the BL translation of any program in one of our mini-languages will have a "prologue" to bind the identifiers of the program. For example, the prologue for an ML-0 ⟨program⟩ whose ⟨identifier⟩s are x, y and z will be the BL macro-instruction .setl (x,y,z), which expands into the sequence <u>create</u> x; <u>create</u> y; <u>create</u> z, creating nodes for the cells bound to these ⟨identifier⟩s. Integer values are represented in the base language model by elementary objects of type integer.

As for the translation rules themselves, we give sample ML-0 statements (⟨assignment⟩s) and the BL code they are translated into. Each example is illustrated by one or two "before and after" pictures showing the change the statement makes in the local structure. Although our examples are meant to be indicative rather than exhaustive, they should be more than sufficient to give the reader a complete picture of the rules for translation from ML-0 into BL.

There are essentially three kinds of ⟨assignment⟩s in ML-0:

(1) ⟨identifier⟩ ← nil

e.g. x ← nil is translated into the BL code

clear x (fig. 3.2-1).

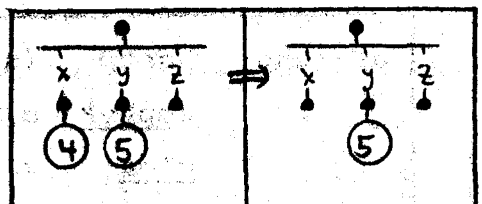


Fig. 3.2-1. Effect of the ML-0 ⟨assignment⟩ x ← nil

(2) ⟨identifier⟩ ← ⟨integer⟩

e.g. y ← 2 is translated into the BL code

const 2,y (figs. 3.2-2 and 3.2-3).

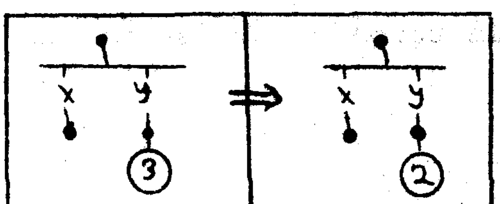


Fig. 3.2-2. Effect of y ← 2 in ML-0

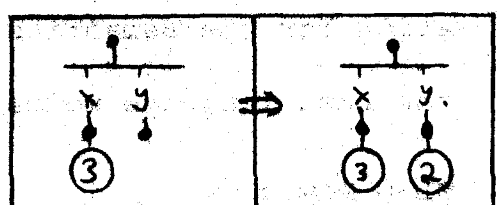


Fig. 3.2-3. Effect of y ← 2 in ML-0

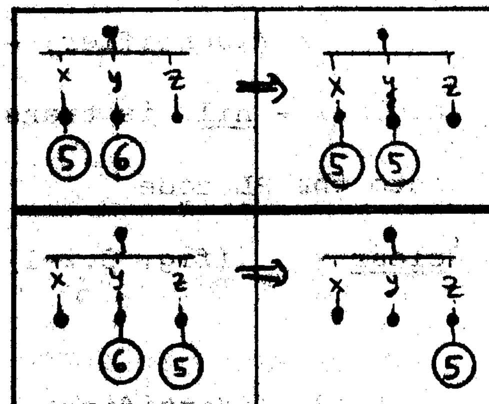
(3) ⟨identifier⟩ ← ⟨identifier⟩

e.g. y ← x is translated into the BL code

.call assign0,(x,y). This code invokes a BL procedure named assign0, which performs the operation specified by the ML-0 ⟨assignment⟩. The definition of the procedure assign0 is shown in figure 3.2-4, and two examples of the ML-0 ⟨assignment⟩ y ← x are pictured in figure 3.2-5.

```
assign0:   .getp    (u,v)
           empty?   u,mov
           clear    v
           return
mov:       const    *u,v
           return
```

Figure 3.2-4. Definition of the BL procedure assign0



Figs. 3.2-5. Effect of y ← x in ML-0

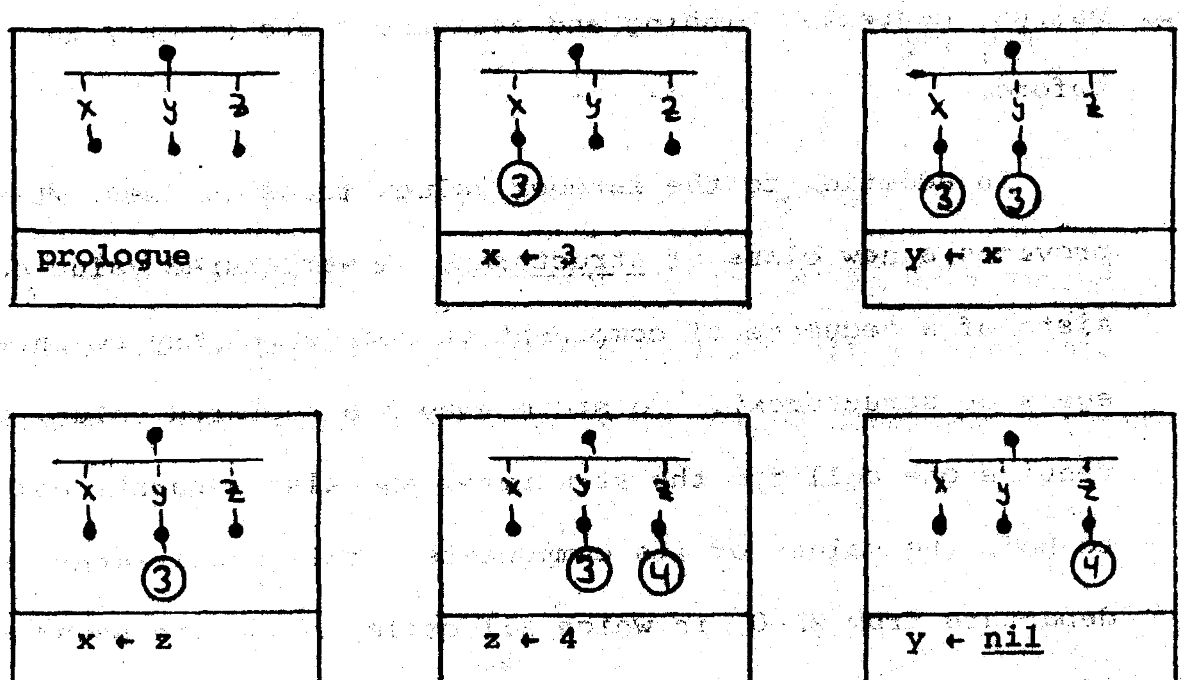
The three translation rules here give us a precise formulation for the semantics of ML-0 in terms of the semantics of the base language model.

ML-0 Movie

We conclude this section by giving a sample ML-0 ⟨program⟩ together with its BL translation. Our example is accompanied by a sequence of pictures forming a "movie" to illustrate the changing state of the local structure as the program is interpreted, statement by statement.

```
ML-0            BL

                .setl   (x,y,z)
x ← 3;          const   3,x
y ← x;          .call   assign0,(x,y)
x ← z;          .call   assign0,(z,x)
z ← 4;          const   4,z
y ← nil         clear   y
```



3.3. Mini-Language 1 -- Structures

Mini-Language 1 (ML-1) adds the notion of data structures to the foundation provided by ML-0. As we have said before, a structure is a data object which consists of individually accessible component objects. There are two fundamental operations relating directly to this concept of structures: (1) construction of a structured object whose components will be objects with given values, and (2) selection of component objects from a structure. ML-1 provides for these operations while retaining intact the concepts and mechanisms of ML-0. In particular, the notions of cells, values, contents, binding and assignment are exactly as before.

In addition to the integer values found in ML-0, ML-1 provides a new class of structures. A structured value consists of a sequence of component values (which may be integers or structures). To store away a structured value, we require one cell for the structure, and also separate cells to hold the values of its components. This requirement is a departure from ML-0, in which all cells in use are bound to identifiers. Component cells must now be handled by some kind of free-storage management technique or cell allocator.

In ML-1, a cell may assume successive values of different types (an integer one moment and a structure the next, or vice versa). There are no restrictions on what values may be stored in which cells. There is a need, however, to detect references to nonexistent components of a structure. Such error-checking will have to be performed by the defining interpreter.

<u>Syntax of ML-1</u>

There is a new primitive syntactic class here, namely ⟨selector⟩, which denotes alphanumeric strings together with integers.

```
⟨program⟩      ::= ⟨assignment⟩ ; ... ; ⟨assignment⟩
⟨assignment⟩   ::= ⟨destination⟩ ← ⟨expression⟩
⟨expression⟩   ::= ⟨destination⟩ | ⟨generator⟩ | nil
⟨destination⟩  ::= ⟨identifier⟩ | ⟨selection⟩
⟨selection⟩    ::= ⟨selector⟩ of ⟨expression⟩
⟨generator⟩    ::= ⟨integer⟩ | ⟨construction⟩
⟨construction⟩ ::= [ ⟨field⟩ ; ... ; ⟨field⟩ ]
⟨field⟩        ::= ⟨selector⟩ : ⟨expression⟩
```

<u>Description</u>

Structures in ML-1 are sequences of component values. Each component in a structure has associated with it a ⟨selector⟩. The <u>selection</u> operation gives individual access to the components of a structure by using the ⟨selector⟩s to indicate the appropriate components. Thus, for example, the ⟨selection⟩ a <u>of</u> x refers to the component of the structure x having the ⟨selector⟩ named "a".

The notion of ⟨destination⟩ is extended in ML-1 to include selections of component objects from structures. In particular, ⟨selection⟩s may appear on both sides of ⟨assignment⟩s. This allows for selective updating of components of a structure. A ⟨selection⟩ occurs as an instance of a ⟨destination⟩ and refers to a component cell for a structure. In this way, ML-1 preserves the ML-0 association between ⟨destination⟩s and cells.

Also as in ML-0, distinct ⟨destination⟩s refer to distinct cells. There is no sharing of data.

All values in ML-1 are created by instances of ⟨generator⟩s. A ⟨construction⟩ is a special kind of ⟨generator⟩ provided by ML-1 for building structured values. In a ⟨construction⟩, we simply supply ⟨expression⟩s yielding values for the components with the associated ⟨selectors⟩. Each component name/value pair is called a ⟨field⟩. Thus the two kinds of ⟨generator⟩s, namely ⟨integer⟩s and ⟨construction⟩s, produce the two kinds of values in ML-1.

Semantics of ML-1 (informal)

As with ML-0, in order to lend precision to the notions we have introduced, we give an informal description of the semantics associated with each significant syntactic class of ML-1.

(1) ⟨program⟩s: The semantic rule for an ML-1 ⟨program⟩ is identical to rule (1) in the previous section for ML-0 ⟨program⟩s.

(2) ⟨assignment⟩s: ML-1 ⟨assignment⟩s work by the same principles as in ML-0, but there is a new factor here. Suppose the value yielded by the ⟨expression⟩ on the right-hand side of an ⟨assignment⟩ is some structure. Then new cells must be allocated to store the component values of this structure. The component cells are said to be subordinate to the cell for the structure they belong to (i.e. to the cell referred to by the ⟨destination⟩ on the left-hand side of the ⟨assignment⟩). Moreover, if a cell containing a structured value is assigned some new value, then the component cells subordinate to this cell are detached and left for the cell allocator to garbage-collect. Structured values are copied on assignment, component by component (and recursively for structure-valued components).

(3) ⟨destination⟩s: There are two kinds of ⟨destination⟩s in ML-1. ⟨identifier⟩s are handled exactly

as in rule (3) for ML-0. We now discuss ⟨selection⟩s.

(4) ⟨selection⟩s: A ⟨selection⟩ consists of a ⟨selector⟩ and an ⟨expression⟩. The value yielded by the ⟨expression⟩ (see rule (5) below) is determined. This value must be a structure, or the effect of the ⟨selection⟩ is undefined. Furthermore, this structure must have some component with the given ⟨selector⟩. Finally, this component must be stored in some component cell (which was allocated when the structured value was constructed). Then this component cell is the cell referred to by the ⟨selection⟩.

(5) ⟨expression⟩s: With respect to the three kinds of ⟨expression⟩s in ML-1, the occurrence of the indicator nil or of a ⟨destination⟩ is treated exactly as in ML-0. As for ⟨generator⟩s, the only aspect we need to explain here is the semantic rule for ⟨construction⟩s.

(6) ⟨construction⟩s: A ⟨construction⟩ consists of a sequence of ⟨field⟩s, each with a ⟨selector⟩ and an ⟨expression⟩. Each ⟨field⟩ represents a component with the indicated ⟨selector⟩ and with value yielded by the ⟨expression⟩. The rule for interpretation of a ⟨field⟩

consists of three steps --

(i) Evaluate its <expression>.

(ii) Allocate a new cell and store the value from step (i) in it (the new cell remains empty if step (i) yields no value).

(iii) Associate the newly allocated component cell (and the value it now contains) with the <selector> of the <field>.

The semantic rule for a <construction> is to interpret its <field>s sequentially, left to right, as specified above. This results in a series of component values stored in component cells and accessible by <selector>s, or, as we better know it, a structure. There is one additional restriction on <construction>s: the <selector>s of its <field>s must be distinct, or else such a <construction> is illegal and has undefined effect.

BL Representation

We represent structures in ML-1 by BL-objects in which the root node corresponds to the cell we store the structure in, and in which the arcs are labeled with the <selector>s of the structure and lead into nodes representing the corresponding component cells. An example we have already seen is the environment (local structure) for a mini-language program, which is a structured value whose <selector>s are the variables used in the program. Another example is the structure generated by the ⟨construction⟩ [a:1; b:[c:2; d:nil]], whose BL representation is pictured in fig. 3.3-1.

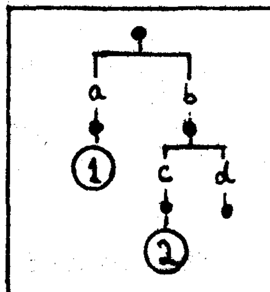


Fig. 3.3-1. BL-object for a structure

A valid ML-1 ⟨destination⟩ corresponds to a node addressable by a compound pathname. For instance, if the structured value of figure 3.3-1 is assigned to the ⟨identifier⟩ x, then the cell referred to by the ⟨destination⟩ c of b of x will be represented by the node x.b.c.

As with ML-0, a ML-1 ⟨program⟩ whose ⟨identifier⟩s are x1, ... , xn has in its BL translation the prologue .setl (x1,...,xn). We now treat translation of various ML-1 ⟨assignment⟩s into BL, illustrating general translation techniques that can be readily applied to any Ml-1 statement. The following cases are representative:

(1) ⟨identifier⟩ ← nil

and (2) ⟨identifier⟩ ← ⟨integer⟩

are both handled exactly as in ML-0 by the respective BL primitives clear and const. Note that the action of these BL instructions disconnects any subordinate component cells that need to be detached.

(3) <identifier> ← <identifier>

e.g. y ← x. This kind of ML-1 <assignment> poses a problem in translation when the source <expression> x has a structured value. In that case, the structured value for x must be copied component by component into y, creating new cells as required to hold new components of y. This kind of action is illustrated in figure 3.2-2. We shall translate the <assignment> y ← x as a call on a BL procedure named assign1, so the BL code for the statement y ← x will be .call assign1,(x,y). The code for the BL procedure assign1 is shown in figure 3.3-3. If x is empty or has an integer value, then assign1 works like the assign0 procedure which translates the corresponding ML-0 <assignment>. If x has a structured value, then for each component of x, we generate a corresponding component for y (allocating a new cell) and call assign1 recursively to give this component

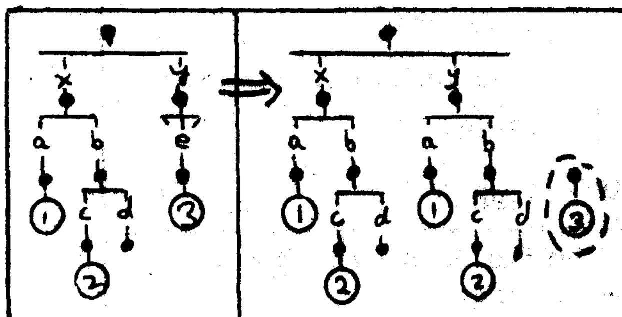


Fig. 3.3-2. Sample effect of the ML-1 <assignment> y ← x when x has structured value.

of y the proper value. Here, the parameter u corresponds

```
assign1: .getp       (u,v)
         clear       v
         nonempty?   u,out
         elem?       u,struc
         const       *u,v
         return
struc:   .getg       (assign1)
loop:    getc        u,i,out
         .call       assign1,(u.*i,v.*i)
         goto        loop
out:     return
```

Figure 3.3-3. Definition of the BL procedure assign1.

to x, and the parameter v corresponds to y.

(4) ⟨identifier⟩ ← ⟨selection⟩

e.g. y ← b *of* x.

The pitfall here is that we must check to verify that x indeed has a b-component. The following BL code takes care of this test:

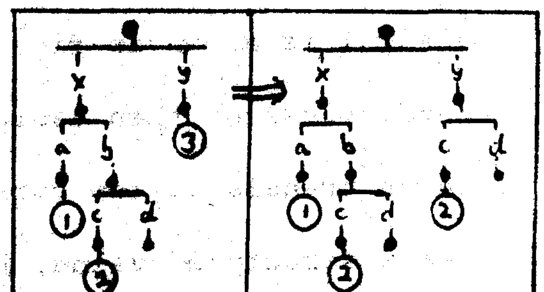


Fig. 3.3-4. Effect of y ← b *of* x in ML-1.

```
has?  x,b,error
.call assign1,(x.b,y)
```

The label "error" refers to some unspecified place we branch to if x has no b-component.

(5) ⟨selection⟩ ← ⟨identifier⟩

e.g. c of a of y ← x is translated into the BL code

```
has?  y,a,error
has?  y.a,c,error
.call assign1,(x,y.a.c)
```

(figure 3.3-5).

(6) ⟨identifier⟩ ← ⟨construction⟩

e.g. y ← [a:3; b:nil; c:x] translates into

```
clear y
const 3,y.a
clear y.b
.call assign1,(x,y.c)
```

(figure 3.3-6).

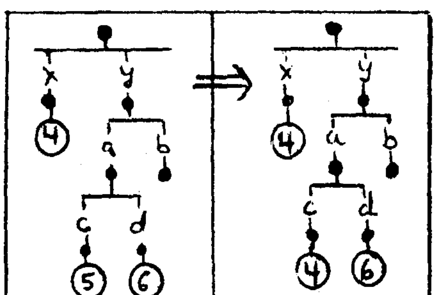


Fig. 3.3-5. Effect of c of a of y ← x

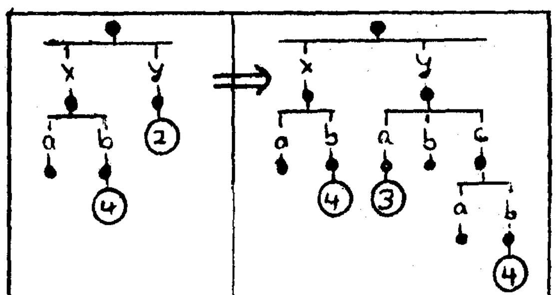


Fig. 3.3-6. Effect of y ← [a:3; b:nil; c:x]

There is a subtle pitfall in these translations. Special care must be taken in translating ⟨assignment⟩s in which the left-hand side and the right-hand side both refer to cells in the same structure. Suppose, for example, that y has the structured value depicted in figure 3.3-7. Translating the ⟨assignment⟩ b <u>of</u> y ← y into the BL code

```
has?  y,b,error
.call assign1,(y,y.b)
```

will not yield the correct results of figure 3.3-8. Instead, there would be a nonterminating sequence of recursive calls of the procedure assign1 (figure 3.3-9). We must therefore translate the

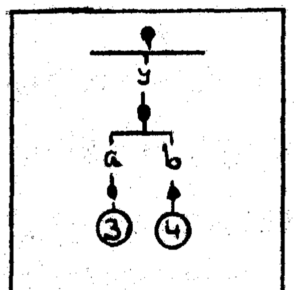


Fig. 3.3-7

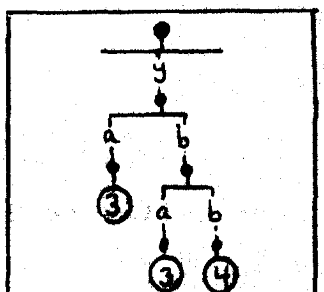


Fig. 3.3-8

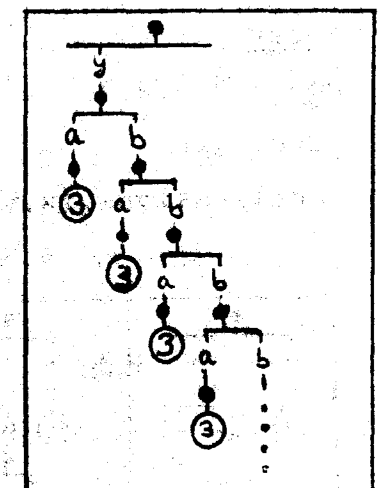


Fig. 3.3-9

⟨assignment⟩ b <u>of</u> y ← y into

```
has?  y,b,error
.call assign1,(y,$temp)
.call assign1,($temp,y.b)
```

With this translation, the recursion terminates because we are not updating the structure $temp during the process of recursively going through its components.

For other cases of "overlapping" assignment, we adopt similar translations. For example, we translate the ⟨assignment⟩ y ← [a:1; b:y] into the BL code

```
.call assign1,(y,$temp)
clear y
const 1,y.a
.call assign1,($temp,y.b);
```

and we translate y ← [c:a of y] into

```
has? y,a,error
clear $temp
link $temp,q,y.a
clear y
.call assign1,($temp.q,y.c).
```

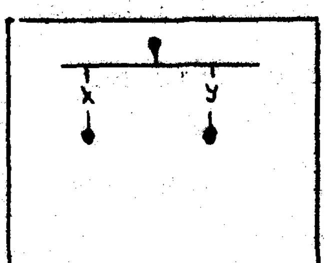
Note that in ML-1, the translator can detect any occurrences of these "overlapping" assignments and make the according adjustments.

ML-1 Movie

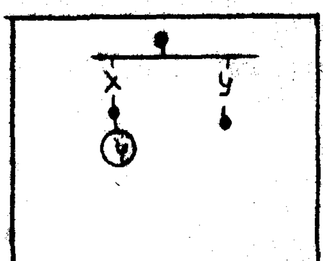
As in the previous section, we conclude with a movie of a sample ML-1 ⟨program⟩ and its translation into BL.

ML-1	BL
	.setl (x,y)
x ← 4;	const 4,x
y ← [a:2; b:x; c:nil];	clear y
	const 2,y.a
	.call assign1,(x,y.b)
	clear y.c

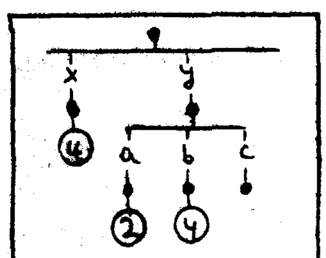
ML-1	BL
x ← a of y;	has? y,a,error
	.call assign1,(y.a,x)
a of y ← 3;	has? y,a,error
	const 3,y.a
x ← y;	.call assign1,(y,x)
y ← [1:a of x;	clear y
2:[r:nil; s:4]];	has? x,a,error
	.call assign1,(x.a,y.1)
	clear y.2
	clear y.2.r
	const 4,y.2.s
s of 2 of y ← a of x;	has? y,2,error
	has? y.2,s,error
	has? x,a,error
	.call assign1,(x.a,y.2.s)
c of x ← x	has? x,c,error
	.call assign1,(x,$temp)
	.call assign1,($temp,x.c)



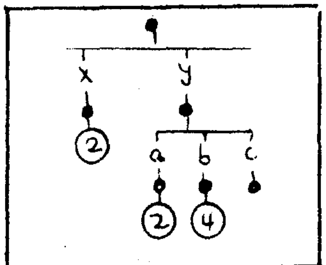
prologue



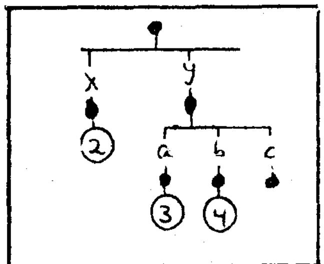
x ← 4



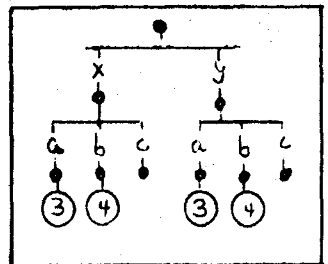
y ← [a:2;b:x;
c:nil]



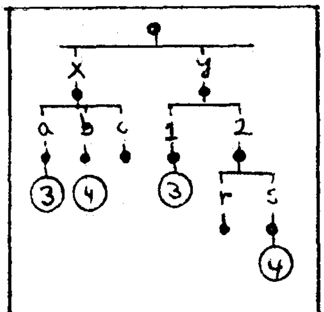
x ← a of y



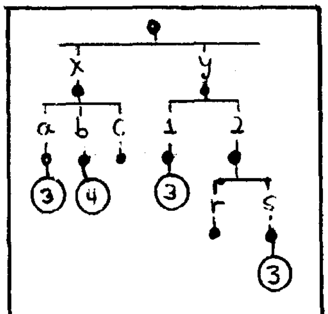
a of y ← 3



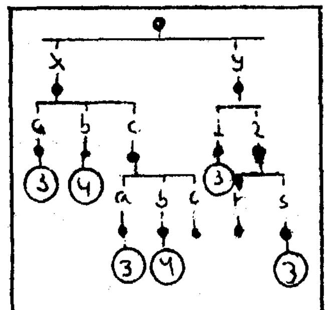
x ← y



y ← [1:a of x;
2:[r:nil;
s:4]]



s of 2 of y
← a of x



c of x ← x

3.4. Mini-Language 2 -- Pointers

Mini-Language 2 (ML-2) extends the concepts we have developed and treats the notion of pointers (references). A pointer is a means by which one can indirectly access a cell and its contents. As with structures, there are two basic operations inherent in the concept of pointers: (1) creation of a pointer value which refers to a given cell, and (2) accessing the cell a pointer "points" to. We wish to provide for these operations while preserving the concepts and mechanisms that have already been developed in this chapter.

In ML-2, there is a new class of pointer values. As with ML-1, cells can accommodate successive values of different classes. We will not, however, allow indirect references through values which are not pointers.

One respect in which the notion of pointer differs from previous concepts is that a pointer value contains information about the cell it refers to. Previous concepts of value had nothing to do with cells. We shall see some of the difficulties caused by this extension.

In this section, we treat ML-2 as an extension of ML-1. However, it is not necessary to include structures in order to handle the new notion of pointers. One could alternatively omit structures from ML-2 and view it as a direct extension to ML-0.

Syntax of ML-2

The "boxed" portion of the ML-2 syntax is that part of ML-2 that deals with structured values and the basic operations on them.

```
⟨program⟩       ::= ⟨assignment⟩ ; ... ; ⟨assignment⟩
⟨assignment⟩    ::= ⟨destination⟩ ← ⟨expression⟩
⟨expression⟩    ::= ⟨destination⟩ | ⟨generator⟩ | nil
⟨destination⟩   ::= ⟨identifier⟩ | ⟨indirect⟩ | ⟨selection⟩
⟨indirect⟩      ::= val ⟨expression⟩
⟨selection⟩     ::= ⟨selector⟩ of ⟨expression⟩
⟨generator⟩     ::= ⟨integer⟩ | ⟨pointer⟩ | ⟨construction⟩
⟨pointer⟩       ::= ptr ⟨destination⟩
⟨construction⟩  ::= [ ⟨field⟩ ; ... ; ⟨field⟩ ]
⟨field⟩         ::= ⟨selector⟩ : ⟨expression⟩
```

Description

There are two new syntactic classes in ML-2. A ⟨pointer⟩, consisting of the symbol ptr and a ⟨destination⟩, specifies the creation of a pointer value which will refer to the same cell as the ⟨destination⟩. The only way to build pointer values in ML-2 is by means of ⟨pointer⟩s; we therefore classify the ⟨pointer⟩ syntactically as an instance of a ⟨generator⟩. An ⟨indirect⟩, consisting of the symbol val and a (pointer-valued) ⟨expression⟩, is ML-2's way of accessing the cell referred to by a pointer value. As such, an ⟨indirect⟩ is a kind of ⟨destination⟩.

We have already seen all the other ML-2 syntax classes.

Semantics of ML-2 (informal)

All we need to give here are informal semantic rules corresponding to the two new syntactic classes. All the other semantic rules for ML-2 are identical to the corresponding rules for ML-0 or ML-1.

(1) <pointer>s: This kind of <expression> contains a <destination> and yields a pointer value which refers to the same cell as the <destination>.

(2) <indirect>s: An <indirect> contains an <expression>. The value yielded by the <expression> is determined. If it isn't a pointer, the <indirect> has undefined value. Otherwise the <indirect> specifies the cell referred to by this pointer value.

BL Representation

Deciding on a way to represent pointer values in BL presents difficulties. In most conventional systems, pointer values are simply the numeric addresses of cells. However, in the base language model, referencing of cells is symbolic. The most straightforward approach to this problem is to view a cell's pathname (i.e. sequence of selectors from the root node of the current local structure) as its address. A pointer value would then be represented in the base language model by an elementary string value encoding the pathname of the cell pointed to. Under such a scheme, after executing the ML-2 instructions

x ← 3; y ← ptr x; z ← y; w ← val y

the environment would appear as in figure 3.4-1. After the further instructions

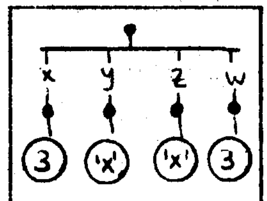


Fig. 3.4-1

z ← x; val y ← ptr z

are executed, the environment would then appear as in figure 3.4-2. Under such a scheme, translation into BL would not be difficult. However, this approach breaks down in the presence of structures. For example, execution of the sequence of ML-2 instructions

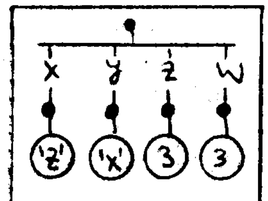


Fig. 3.4-2

x ← [a:2]; y ← ptr a of x

would result in y having as value the pathname "x.a" (figure 3.4-3). If we then execute the ⟨assignment⟩ x ← 3, x would no longer have an a-component; the cell containing the value 2 would therefore no longer have the pathname x.a and would hence be inaccessible through y. In other words, under this

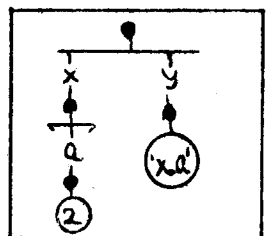


Fig. 3.4-3

scheme there is no way to provide for retention of cells referred to by pointers. The main conceptual weakness of this scheme is that the address of a cell depends on a particular path of access to it. Such a dependence is to be avoided.

A second way to refer to a cell is by directly linking to it, that is, sharing it. It is imperative that the pointer have a separate cell for itself as well as the cell it points to. Otherwise, after executing the ML-2 instructions x ← 3; y ← ptr x we would have a situation as pictured in figure 3.4-4 in which the ⟨assignment⟩ y ← 2 would erroneously affect x (we want to access x through y only by use of the ⟨indirect⟩ val y). To insure separate cells, we will make a pointer value an instance of a structure, where the cell pointed to will be the sole component cell. Thus the result of executing the instructions

x ← [a:2]; y ← ptr a of x

will be as in figure 3.4-5, and after the further instruction x ← 3, we see that the cell containing the value 2 is proper-

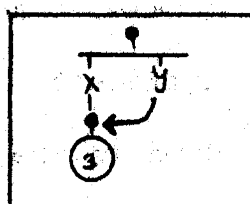


Fig. 3.4-4

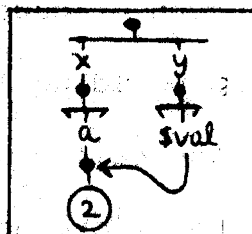


Fig. 3.4-5

ly retained (figure 3.4-6). Note that we have adopted the reserved name "$val" as the selector for the single component of an ML-2 pointer value under our representation scheme (to avoid clashes with the ⟨selector⟩s of ML-2 structures).

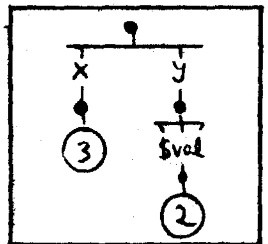


Fig. 3.4-6

Now that we have settled on a BL representation for pointer values, translation of ML-2 into BL is straightforward. We only need consider four new cases of ⟨assignment⟩s:

(1) ⟨identifier⟩ ← ⟨pointer⟩

e.g. y ← <u>ptr</u> x is translated into the BL code

<u>clear</u> y
<u>link</u> y,$val,x

(2) ⟨identifier⟩ ← ⟨identifier⟩

e.g. y ← x is translated into the invocation .call assign2,(x,y), where the definition of the BL procedure assign2 is shown in figure 3.4-7. The difference between assign1 and assign2 is that assign2 has additional code to handle assignment of pointer values, preventing us from attempting to copy the contents of a cell referred to by some pointer. An example of the assigning of a pointer value is depicted in figure 3.4-8.

```
assign2:  .getp       (u,v)
          clear       v
          nonempty?   u,out
          elem?       u,comp
          const       *u,v
          return
comp:     has?        u,$val,struc
          link        v,$val,u.$val
          return
struc:    .getg       (assign2)
loop:     getc        u,i,out
          .call       assign2,(u.*i,v.*i)
          goto        loop
out:      return
```

Figure 3.4-7. Definition of the BL procedure assign2.

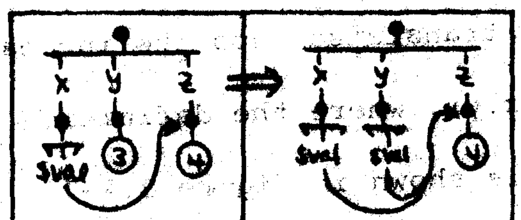


Fig. 3.4-8. Effect of the ML-2 ⟨assignment⟩ y ← x when x has a pointer value.

(3) ⟨identifier⟩ ← ⟨indirect⟩

e.g. z ← val y is translated into the BL code

```
has?  y,$val,error
.call assign2,(y.$val,z)
```

(4) ⟨indirect⟩ ← ⟨expression⟩

e.g. val x ← 3 is translated into the BL code

```
has?  x,$val,error
const 3,x.$val
```

Using these translation schemes, it is easy to produce BL code corresponding to any ML-2 ⟨program⟩. However, the presence of "overlapping" assignments can no longer always be detected by the translator. For example, in the state depicted in figure 3.4-9, we want the ⟨assignment⟩ b of y ← val x to result in the state shown in figure 3.4-10. The BL code

```
has?  y,b,error
has?  x,$val,error
.call assign2,(x.$val,
               $temp)
.call assign2,($temp,
               y.b)
```

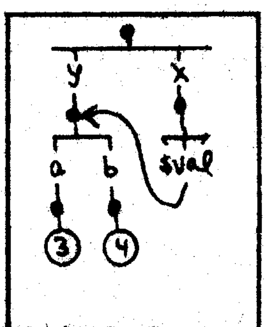


Fig. 3.4-9

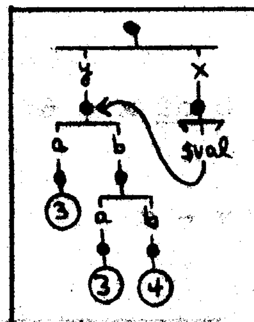
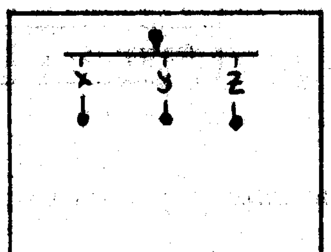


Fig. 3.4-10

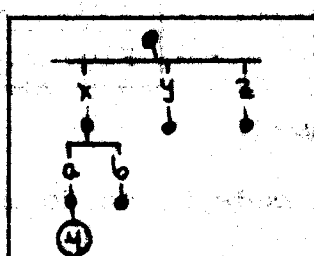
works properly. In other words, the translator must produce BL code to perform extra copying whenever there is a possibility of overlap. This is a major source of inefficiency, since overlap is probably an infrequent event.

ML-2 Movie

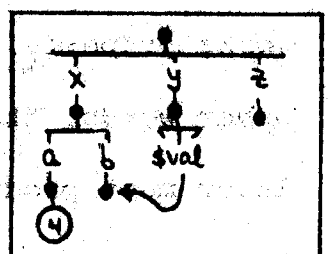
ML-2	BL
	.setl (x,y,z)
x ← [a:4; b:nil];	clear x
	const 4,x.a
	clear x.b
y ← ptr b of x;	has? x,b,error
	clear y
	link y,$val,x.b
val y ← 5;	has? y,$val,error
	const 5,y.$val
z ← [c:y; d:val y; e:ptr z];	has? y,$val,error
	.call assign2,(y.$val,$temp)
	clear z
	.call assign2,(y,z.c)
	.call assign2,($temp,z.d)
	link z.e,$val,z
b of x ← 6;	has? x,b,error
	const 6,x.b
x ← z	.call assign2,(z,x)



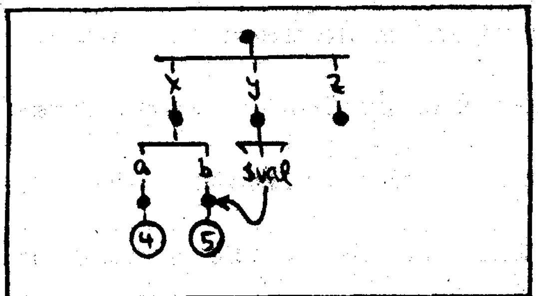
prologue



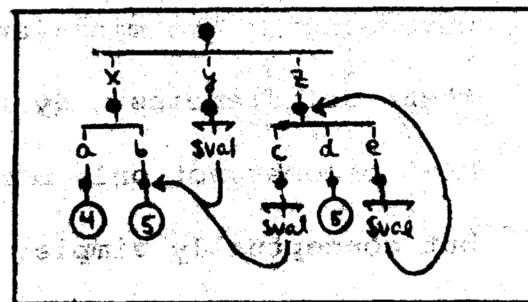
x ← [a:4; b:nil]



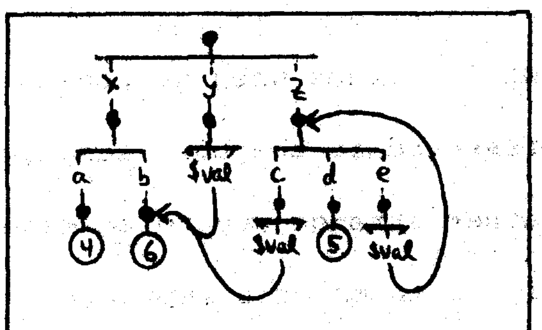
y ← ptr b of x



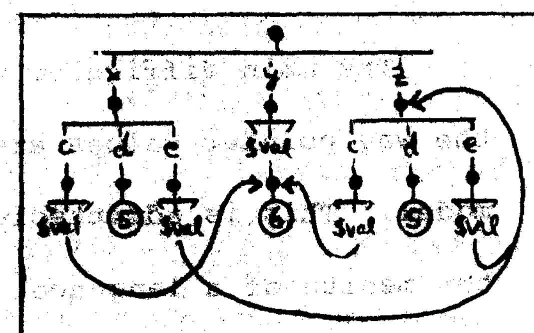
<u>val</u> y ← 5



z ← [c:y; d:<u>val</u> y;
e: <u>ptr</u> z]



b <u>of</u> x ← 6



x ← z

3.5. Mini-Language 3 -- Sharing

So far in this chapter, we have progressed through three mini-languages in developing our semantic model for data structures and pointers. Although ML-2 handles all of these concepts, there are some respects in which the design we so carefully built up becomes cumbersome and inelegant. In this section we shall look at some of the weaknesses of ML-2 and see how they reflect a conceptual shortcoming in our design. The mini-language ML-3 is devised to remedy these deficiencies. By revising the notion of structures, ML-3 becomes not only more powerful and efficient than ML-2, but conceptually simpler as well. In fact, the entire apparatus of pointers that was developed in the previous section is subsumed within the re-definition of structured value.

The main difficulty with ML-2 emerges when we consider the way pointer values are represented in the base language model. This is admittedly a rather strange way to examine the merits of a language, namely in terms of a representation decision with respect to a particular semantic model. But the base language model is special in that it was specifically designed for the purpose of describing the concepts of sharing which we are studying. So it is perfectly valid to use insights provided by this model to aid in designing mini-languages which deal with data structures and sharing.

In the last section, we chose to represent a pointer value in the base language model as a one-component structure whose component cell is precisely the cell pointed to. In other words, pointer values are instances of structures whose components share with other data objects. It is this much more general concept of shared data objects that concerns us in this section. The only kind of sharing provided in ML-2 is the pointer, which is a structure having exactly one component cell, shared with some object. In the course of trying to model aspects of real-world programming languages in ML-2, this limitation becomes a stumbling block. For example, the notion of <u>tuple</u> in languages like BASEL is that of a vector of addresses, i.e. a structure with an <u>arbitrary</u> number of components sharing with other objects. In ML-2, this can be modeled only as a structure whose components are pointers. These components, when represented in the base language model, take up an extra level of indirection, which becomes a bit clumsy.

To give a better treatment to this generalized notion of sharing, we revise our concept of structure. In ML-2, as in ML-1, the notion of structured values as being composed of components with ⟨selector⟩s and values does not directly utilize the concept of cells. Cells are part of only pointer values. What we've done in ML-2 is represent pointers like structures but use a different set of rules to manipulate them. This conceptual distinction puts the two notions -- structured values and pointer values -- almost at odds with each other in ML-2. We include cells in our revised concept of structured values in ML-3; as a result of this, the need for a separate class of pointer values vanishes.

A structured value in ML-1 and in ML-2 was a collection of components, each consisting of a value and an associated ⟨selector⟩. In ML-3, we define a component of a structure to now be a ⟨selector⟩-*cell* pair, rather than a ⟨selector⟩-*value* pair. The value of a structured object is still the set of its components.

Syntax of ML-3

```
⟨program⟩        ::= ⟨assignment⟩ ; ... ; ⟨assignment⟩
⟨assignment⟩     ::= ⟨destination⟩ ← ⟨expr⟩
⟨expr⟩           ::= ⟨destination⟩ | ⟨generator⟩
                        | ⟨modification⟩ | nil
⟨destination⟩    ::= ⟨identifier⟩ | ⟨selection⟩
⟨selection⟩      ::= ⟨selector⟩ of ⟨expr⟩
⟨generator⟩      ::= ⟨integer⟩ | ⟨construction⟩
⟨construction⟩   ::= [ ⟨field⟩ ; ... ; ⟨field⟩ ]
⟨field⟩          ::= ⟨selector⟩ : ⟨cell expr⟩
⟨cell expr⟩      ::= share ⟨destination⟩ | ⟨expr⟩
⟨modification⟩   ::= ⟨construction⟩ ⟨expr⟩
```

Description

The syntactic classes of ML-3 are identical to those of ML-1, with two additions. First, there are now two kinds of expressions in ML-3: an ⟨expr⟩ yields a value, and a ⟨cell expr⟩ yields a cell. The only occurrence of ⟨cell expr⟩s is within the ⟨field⟩s of a ⟨construction⟩ (where there used to be ⟨expr⟩s in ML-1 and ML-2). The rules for evaluating both kinds of expressions are given below. The second addition is a new kind of ⟨expr⟩, namely the ⟨modification⟩ which yields structured objects built from other structures. All other syntactic classes are exactly as they were in ML-1.

Semantics of ML-3 (informal)

The semantic rules for ⟨program⟩s, ⟨assignment⟩s, ⟨destination⟩s, ⟨identifier⟩s and ⟨selection⟩s are identical to the rules given for ML-1. The remaining elements warrant some discussion.

(1) ⟨expr⟩s: The occurrence of nil or of a ⟨destination⟩ as an ⟨expr⟩ is handled just as in ML-0 and ML-1. ⟨generator⟩s are either ⟨integer⟩s, which are handled as before, or ⟨construction⟩s, which are described in

rule (2) below. ⟨modification⟩s are discussed in rule (6) below.

(2) ⟨construction⟩s: The semantics of ⟨constructions⟩ and ⟨field⟩s follows directly from the new ML-3 notion of structures. A ⟨construction⟩ denotes the value of a structure which is generated on the spot. A ⟨construction⟩ consists of a series of ⟨field⟩s, each with a ⟨selector⟩ and a ⟨cell expr⟩. Each ⟨field⟩ represents a component consisting of this ⟨selector⟩ and the cell yielded by the ⟨cell expr⟩ (see rule (3) below). Finally, the structured value yielded by the ⟨construction⟩ is the set of components given by its ⟨field⟩s. We make one restriction on ⟨construction⟩s: the ⟨selector⟩s of its ⟨field⟩s must be distinct, or else the ⟨construction⟩ is invalid and has undefined effect.

(3) ⟨cell expr⟩s: The two kinds of ⟨cell expr⟩ are discussed in rules (4) and (5) below.

(4) shared ⟨destination⟩s: A ⟨cell expr⟩ of the form share ⟨destination⟩ yields the cell referred to by the ⟨destination⟩. This is the basic source of sharing in ML-3; shared ⟨destination⟩s are used to build structures having components whose cells are already in use. It is this facility which subsumes the ML-2 notion of pointers.

(5) *⟨expr⟩s as ⟨cell expr⟩s*: The cell yielded by an ⟨expr⟩ occurring as a ⟨cell expr⟩ is a newly-allocated cell distinct from all cells in use and containing the value yielded by the ⟨expr⟩. Evaluation of a ⟨cell expr⟩ of form ⟨expr⟩ is the only way to allocate new cells in ML-3.

(6) *⟨modification⟩s*: A ⟨modification⟩ consists of a ⟨construction⟩ and an ⟨expr⟩. The value of the ⟨expr⟩ (which we call the *modificand*) must be a structure or the indicator *nil*, or else the effect of the ⟨modification⟩ is undefined. The value yielded by the ⟨modification⟩ will be a newly-generated structure whose components are obtained as follows:

(i) Each component of the modificand whose ⟨selector⟩ belongs to no ⟨field⟩ of the ⟨construction⟩ will be a component of the new structure.

(ii) For each ⟨field⟩ of the ⟨construction⟩ there will be in the new structure a component with the same ⟨selector⟩ and as its cell the cell yielded by the ⟨cell expr⟩ of the ⟨field⟩.

Alternatively, we can view each ⟨field⟩ of the ⟨construction⟩ as either replacing or appending a component to the modificand depending on whether or not its ⟨selector⟩ belongs to some component of the modificand. Note that evaluation of a ⟨modification⟩ may cause allocation of new cells, but it does not in any way affect the contents of existing cells. Strictly speaking, ⟨modification⟩s are redundant in ML-3. If, for example, the ⟨identifier⟩ x has a structured value with two components whose ⟨selector⟩s are a and b, then the ⟨modification⟩ [b:3; c:share y] x will yield the same value as the ⟨construction⟩ [a:share a of x; b:3; c:share y].

BL Representation

We represent a structured value in ML-3 by a BL-object whose arcs lead into the nodes for the component cells and are labeled with the corresponding ⟨selector⟩s. This is straightforward, simple and clean.

We now give sample translations of ML-3 ⟨assignment⟩s into BL.

(1) ⟨identifier⟩ ← nil

and (2) ⟨identifier⟩ ← ⟨integer⟩

are both handled as in ML-0 and ML-1.

(3) ⟨identifier⟩ ← ⟨identifier⟩

e.g. y ← x is translated into .call assign3 (x,y), where the BL procedure assign3 is defined in figure 3.5-1. The code is the same as for the procedure assign1 for empty and integer values of the source ⟨identifier⟩ x, except for the

the presence of the same? x,y,out test which makes sure the ⟨assignment⟩ is nontrivial (otherwise the clear instruction would destroy the value we want to keep). If x has a structured value, then y will get the same structured value. This means, by the new definition of structured value, that the components of y will now share with the components of x (figure 3.5-2). In executing any ⟨assignment⟩,

```
assign3: .getp      (u,v)
         same?      u,v,out
         clear      v
         nonempty?  u,out
         elem?      u,struc
         const      *u,v
         return
struc:   getc       u,i,out
         link       v,*i,u.*i
         goto       struc
out:     return
```

Fig. 3.5-1. Definition of the BL procedure assign3

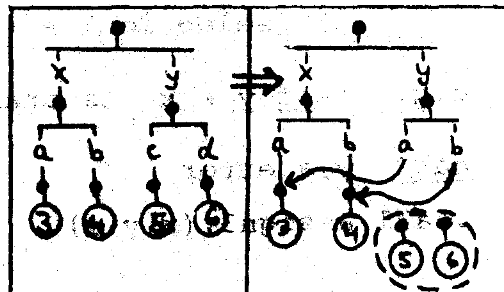


Fig. 3.5-2. Effect of the ML-3 ⟨assignment⟩ y ← x when x has a structured value

the contents of exactly one cell will be copied. Component cells are now shared, not copied. Note that this is a vast gain in efficiency for ML-3 over ML-1 and ML-2. The "meaning" of the ⟨assignment⟩ y ← x, then, differs between ML-1 and ML-3. For example, after executing the instructions x ← [a:3; b:4]; y ← x; a <u>of</u> y ← 5, then the expression a <u>of</u> x will yield the value 3 in ML-1 (and ML-2), but will evaluate to 5 in ML-3.

(4) ⟨identifier⟩ ← ⟨selection⟩

e.g. y ← b <u>of</u> x is translated into the BL code

```
has?  x,b,error
.call assign3,(x.b,y)
```

(5) ⟨selection⟩ ← ⟨identifier⟩

e.g. a <u>of</u> y ← x is translated into the BL code

```
has?  y,a,error
.call assign3,(x,y.a)
```

(6) ⟨identifier⟩ ← ⟨construction⟩

e.g. y ← [c:x; d:b <u>of</u> x; e:<u>share</u> z] is translated into

```
has?  x,b,error
.call assign3,(x.b,
              $temp)
clear y
.call assign3,(x,y.c)
.call assign3,($temp,
              y.d)
link  y,e,z
```

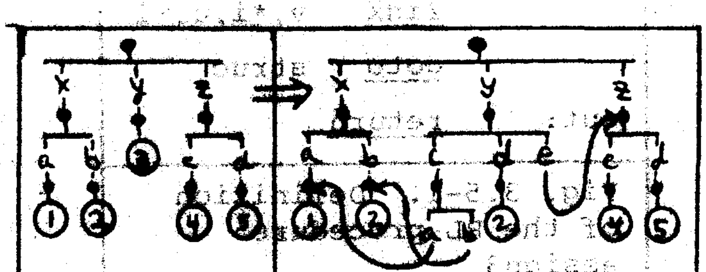


Fig. 3.5-3. Effect of y ← [c:x; d:b <u>of</u> x; e:<u>share</u> z] in ML-3

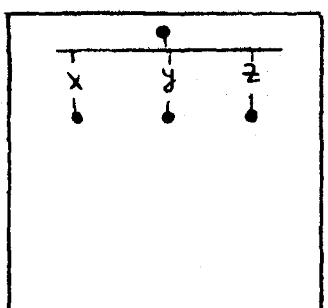
Note that overlapping ⟨assignment⟩s pose no problem at all for statements of types (4) and (5). This is due to the fact that component cells of a structure are no longer copied on assignment. However, we do need the use of temporaries in ⟨assignment⟩s involving ⟨construction⟩s, for instance, to take care of the case when y shares with b <u>of</u> x before executing the ⟨assignment⟩ in example (6) above.

Finally, we note that pointers in ML-2 have been subsumed in ML-3. In place of the ML-2 <u>ptr</u> ⟨destination⟩ we can write the ML-3 ⟨construction⟩ [val:share ⟨destination⟩], and wherever ML-2 uses <u>val</u> ⟨expr⟩, ML-3 substitutes val <u>of</u> ⟨expr⟩.

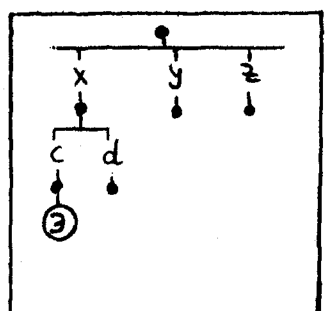
<u>ML-3 Movie</u>

<u>ML-3</u>	<u>BL</u>	
	.setl	(x,y,z)
x ← [c:3; d:<u>nil</u>];	<u>clear</u>	x
	<u>const</u>	3,x.c
	<u>clear</u>	x.d
z ← [a:4; b:[q:c <u>of</u> x;	<u>has?</u>	x,c,error
r:<u>nil</u>]];	.call	assign3,(x.c,$temp)
	<u>clear</u>	z
	<u>const</u>	4,z.a
	<u>clear</u>	z.b
	.call	assign3,($temp,z.b.q)
	<u>clear</u>	z.b.r

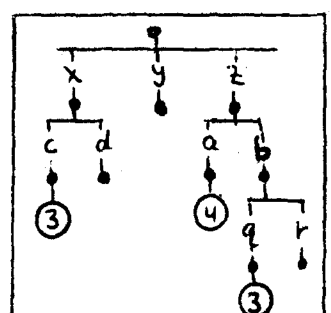
ML-3	BL
y ← [p:share x];	clear y
	link y,p,x
p of y ← y;	has? y,p,error
	.call assign3,(y,y.p)
y ← b of z;	has? z,b,error
	.call assign3,(z.b,y)
x ← [b:5] z;	.call assign3,(z,x)
	const 5,x.b
z ← [c:share q of y] z;	has? y,q,error
	link z,c,y.q
y ← [a:b of z; c:share z] x;	has? z,b,error
	.call assign3,(z.b,$temp)
	.call assign3,(x,y)
	.call assign3,($temp,y.a)
	link y,c,z



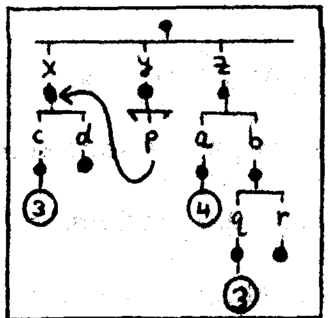
prologue



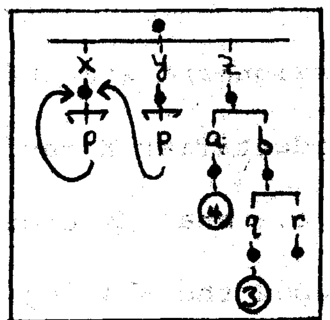
x ← [c:3;
d:nil]



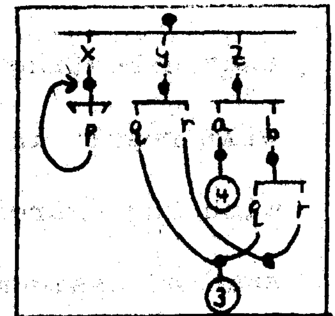
z ← [a:4;
b:[q:c of x;
r:nil]]



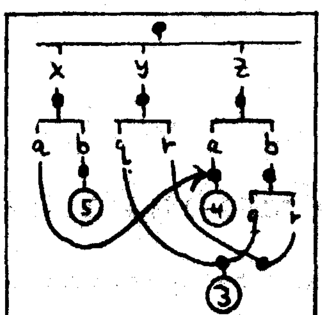
y ← [p:share x]



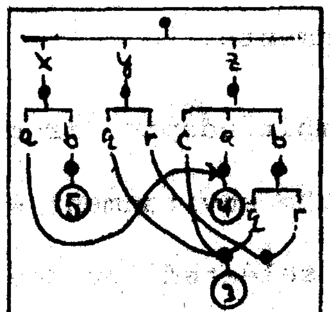
p of y ← y



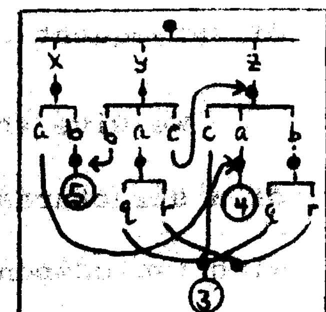
y ← b of z



x ← [b:5] z



z ← [c:share q of y]z



y ← [a:b of z; c:share z]x

3.6. Discussion and Examples

In this chapter we have built up a hierarchy of mini-languages, culminating in ML-3. We now relate this development to the main issues that were raised in Chapter 1. A major concern with respect to a given "real-world" programming language is the effect of its assignment operation on an environment containing structured data objects. We know that executing an assignment statement of the form X := e will result in the identifier X having the value associated with the expression e. What is uncertain is the effect of such an assignment upon the sharing relationships among the various cells in the environment. Variations in sharing properties can in general induce differences in the effect of subsequent assignments.

We give an example adapted from [Bur 68]. The only data structures in the environment will be LISP-like lists with two components selected by the respective selectors head and tail. Burstall compares analogous programs in two languages: List-Algol, which combines ALGOL 60 assignment with structures essentially equivalent to LISP lists, and ISWIM ("If you See What I Mean"), which is based on the same functional lambda-calculus notions as LISP. In both languages, the two-argument function cons returns a list whose head is the first argument and whose tail is the second argument; the functions head and tail select the components from a list. Burstall's two programs are shown in figure 3.6-1. Program A, we are told, prints 3 while program B prints 1 "since it does not cater for the side-effect on y of the assignment to x." This explanation gives little insight into why there should be such a difference in the first place. The obvious distinction between the two programs

	Program A: List-Algol	Program B: ISWIM
1	begin list x,y;	print let x=undef and y=undef;
2	x := CONS(1,nil);	let x = cons(1,nil);
3	y := CONS(2,x);	let y = cons(2,x);
4	HEAD(x) := 3;	let x = cons(3,tail(x));
5	print(HEAD(TAIL(Y)))	result head(tail(y))
	end	

Fig. 3.6-1. Two sample programs with different effects.

lies in line 4. ISWIM, being a functional applicative language, has no direct counterpart to the List-Algol component update statement HEAD(x) := 3. But this is not the root of the semantic difference between the two programs. Burstall neglects to say that even if we change line 4 in Program A to x := CONS(3,TAIL(x)), Program A will still print 3.

The source of the trouble lies in a subtle difference between the cons functions in the two languages. We can pinpoint the distinction by translating both programs into ML-3. Line 2 in both programs can be translated into x ← [head:1; tail:nil], with the resulting environment as in figure 3.6-2. Line 3 in Program A is equivalent to the

ML-3 statement y ← [head:2; tail:<u>share</u> x], while line 3 in Program B is equivalent to y ← [head:2; tail:x]. The respective results are shown in figures 3.6-3 and 3.6-4.

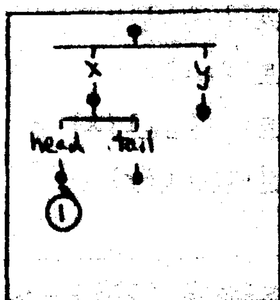


Fig. 3.6-2. State after line 2.

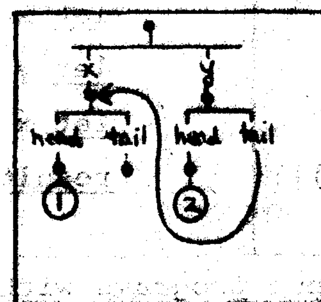


Fig. 3.6-3. After line 3, Program A.

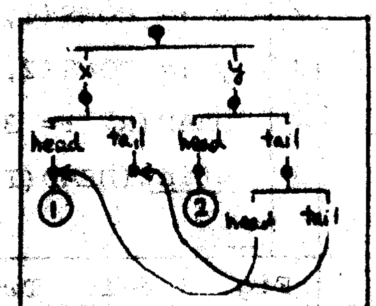


Fig. 3.6-4. After line 3, Program B.

Finally, the <u>revised</u> line 4 for Program A, which reads x := CONS(3,TAIL(x)), is equivalent to the ML-3 statement x ← [head:3; tail:<u>share</u> tail <u>of</u> x], while line 4 of Program B is equivalent to x ← [head:3; tail:tail <u>of</u> x]. The respective results are shown in figures 3.6-5 and 3.6-6.

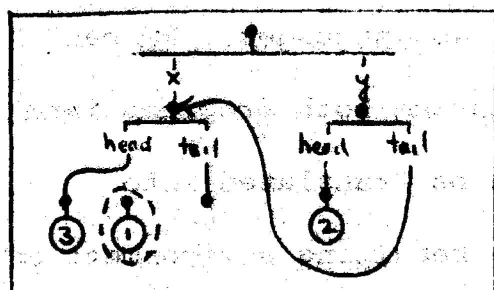


Fig. 3.6-5. After new line 4, Program A.

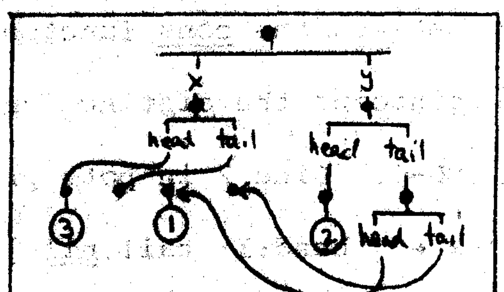


Fig. 3.6-6. After line 4, Program B.

We can see that the ML-3 expression head of tail of y yields 3 in figure 3.6-5 and 1 in figure 3.6-6.

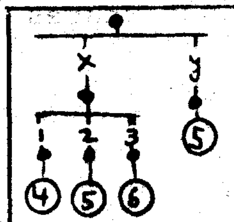
The difference between the two cons functions in Burstall's two languages should now be clear. If an argument to cons is a constant or nil, both languages specify allocation of a new cell to contain the argument value. But if an argument is some identifier, the Lisp-Algol CONS yields for the corresponding component the argument's location, while the ISWIM cons yields the argument's value. This property of the ISWIM cons function is not explicitly stated in Landin's descriptions of ISWIM [Lan 64, Lan 65, Lan 66a]. In fact, the only place from which this property could be readily ascertained was in Burstall's statement that Program B prints the value 1. The ML-3 code into which we translated the statements of the two programs was determined only from the stated results of those programs. What is to be concluded from this is not that Landin was sloppy or vague in his language design and definition, but rather that the language definition methods which are so widely used make it extremely difficult to extract some of the properties of significant practical importance. In other words, a language which features data structures will be better understood and better specified if it defines these facilities in some manner which makes clear the specific sharing relationships among locations.

In the remainder of this section we shall use our mini-languages to talk about the data structuring facilities and mechanisms of several additional programming languages.

PAL

The language PAL [Ev 70] supports only one kind of data structure: the *tuple*. A tuple is a structure whose selectors are consecutive integers starting with 1. As with ML-3, the cell in which a component of a tuple is stored is considered an integral part of the value of the tuple. The PAL expression 4,5,6 specifies the construction of a tuple whose components have the respective values 4,5, and 6; as such, it is equivalent to the ML-3 ⟨construction⟩ [1:4; 2:5; 3:6]. Selection in PAL is expressed by juxtaposition; if the tuple value 4,5,6 is assigned to the variable x, then the PAL expression x 2 evaluates to 5 (it selects the second component). This expression corresponds to the ML-3 ⟨selection⟩ 2 *of* x. The correspondences we have established are summarized in figure 3.6-7.

The concepts of value of a tuple in PAL and value of a structure in ML-3 are very close, and we might expect similar assignments to behave similarly. This is indeed the case, as figure 3.6-8 confirms.



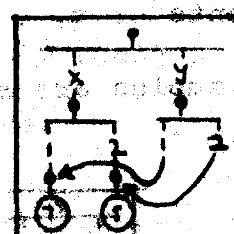
PAL

```
x := 4,5,6;
y := x 2
```

ML-3

```
x←[1:4;2:5;3:6];
y ← 2 of x
```

Fig. 3.6-7. Construction and selection in PAL.



PAL

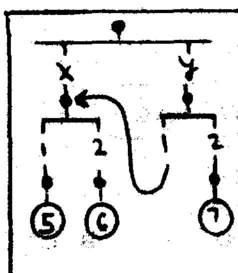
```
x := 7,8;
y := x
```

ML-3

```
x←[1:7;2:8];
y ← x
```

Fig. 3.6-8. Value of a tuple in PAL

PAL has a semantic rule that components of a tuple share with the items in the list expression that constructs it; an example of this rule is shown in figure 3.6-9. This sharing can be blocked using the PAL <u>unshare</u> operator ("$"). Figure 3.6-10 gives an example of this.



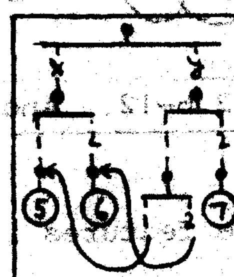
PAL

```
x := 5,6;
y := x,7
```

ML-3

```
x ← [1:5; 2:6];
y ← [1:share x;
     2:7]
```

Fig. 3.6-9. Sharing in PAL tuple construction.



PAL

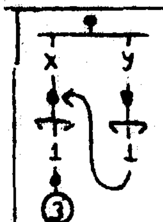
```
x := 5,6;
y := $x,7
```

ML-3

```
x ← [1:5;2:6];
y ← [1:x;2:7]
```

Fig. 3.6-10. Blocking of sharing in PAL.

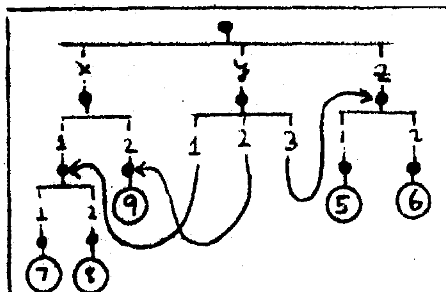
We discuss one more feature of PAL: the aug function. If t is an n-tuple (i.e. tuple with selectors 1,2,...,n) and e is any expression, then the PAL expression t aug e denotes an (n+1)-tuple whose first n components share with the components of t, and whose (n+1)-st component shares with e. Examples are shown in figures 3.6.11 and 3.6.12.



PAL
```
x := nil aug 3;
y := nil aug x
```

ML-3
```
x ← [1:3] nil;
y ← [1:share x] nil
```

Fig. 3.6-11. Example of the use of the PAL function aug



PAL
```
x := (7,8),9;
z := 5,6;
y := x aug z
```

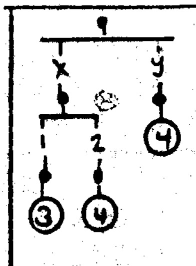
ML-3
```
x ← [1:[1:7;2:8];2:9];
z ← [1:5;2:6];
y ← [3:share y] x
```

Fig. 3.6-12. Another example of aug in PAL

The above features illustrate nearly all of PAL's data structuring capabilities, and they are easily expressed in ML-3. Even though the data-structure facilities of PAL bear a strong resemblance to ML-3, we have given a demonstration of a full-scale, real-world programming language whose data structuring mechanisms have been successfully treated within our model. We discuss two more languages.

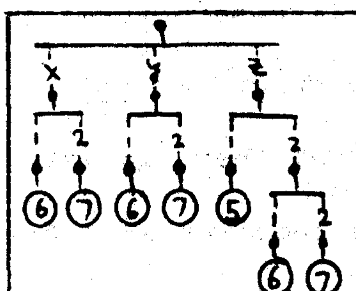
<u>QUEST</u>

The language QUEST [Fenn 73] provides data structures called <u>lists</u> that appear very much like PAL's tuples (see figure 3.6-13). However, the definition of assignment in QUEST treats lists as special cases for which special rules apply. This reduces, essentially, to a treatment of lists in the way ML-1 treats structures. Component values are copied on assignment rather than shared. Figure 3.6-14 presents an example. Note that componentwise copying is coded in ML-3



x ← 3,4; y ← x(2)	QUEST
x := 3,4; y := x 2	PAL
x ← [1:3;2:4]; y ← 2 <u>of</u> x	ML-3

Fig. 3.6-13. Lists in QUEST.

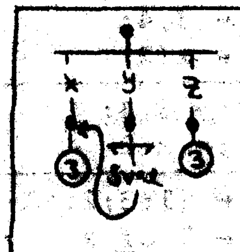


x ← 6,7; y ← x; z ← 5,x	QUEST
x←[1:6;2:7]; y ← x; z←[1:5;2:x]	ML-1
x ← [1:6;2:7]; y ← [1:<u>nil</u>;2:<u>nil</u>]; 1 <u>of</u> y ← 1 <u>of</u> x; 2 <u>of</u> y ← 2 <u>of</u> x; z←[1:5; 2:[1:<u>nil</u>;2:<u>nil</u>]]; 1 <u>of</u> 2 <u>of</u> z ← 1 <u>of</u> x; 2 <u>of</u> 2 <u>of</u> z ← 2 <u>of</u> x	ML-3

Fig. 3.6-14. Copying of components in QUEST assignment

by repeated component updates, reflecting a lack of efficiency. QUEST assignments, unlike their counterparts in PAL, cannot be directly translated into ML-3 without knowing run-time values (i.e. exactly what components a structured value possesses at any given time, so they can be individually updated).

Like ML-2, QUEST handles sharing entirely by means of pointers (called <u>references</u>). Their use is illustrated in figure 3.6-15. There is no appreciable difference between the behavior of these pointers and those in ML-2. Translation into ML-3 would be trivially easy.



QUEST

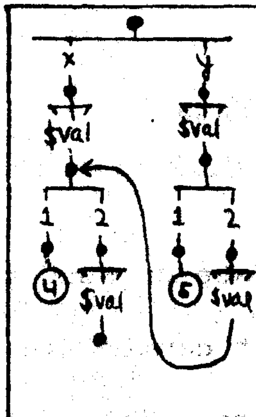
x ← 3;
y ← <u>ref</u> x;
z ← <u>at</u> y

ML-2

x ← 3;
y ← <u>ptr</u> x;
z ← <u>val</u> y

Fig. 3.6-15. References in QUEST.

For the interested reader, the paper on QUEST [Fenn 73] specifies a way to express general ML-3-like structures in QUEST using lists and references. QUEST functions <u>cons</u>, <u>car</u> and <u>cdr</u> are defined, and it is claimed that they simulate their LISP counterparts. The simulation requires an extra level of indirection throughout, a major inefficiency (fig. 3.6-16). Thus we see that using our mini-languages, we have not only able to illustrate the data structuring semantics of QUEST, but we have also perceived a shortcoming in the design of QUEST: like ML-2, QUEST fails to recognize the fundamental significance of the concept of sharing.



QUEST
```
x ← cons(4,nil);
y ← cons(5,x)
```

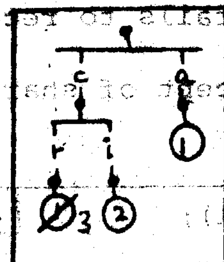
ML-2
```
temp1 ← nil;
temp2 ← [1:4;2:ptr temp1];
x ← ptr temp2;
temp3 ← [1:5;2:ptr val x];
y ← ptr temp3
```

ML-3
```
x←[val:[1:4;2:[val:nil]]];
y←[val:[1:5;
     2:[val:share val of x]]]
```

Fig. 3.6-16. QUEST simulation of LISP <u>cons</u>

SNOBOL4

In the language SNOBOL4 [Gris 71], one finds data structures called "programmer-defined data types." An invocation of the function DATA causes selector and constructor functions to be defined. For example, the invocation DATA('COMPLEX(R,I)') defines the constructor function COMPLEX and the associated selector functions R and I, setting up the correspondence depicted in figure 3.6-17. Beyond this aspect, in which these SNOBOL structures behave exactly as do all the structures we have seen in other languages, the sharing relationships need to be considered.



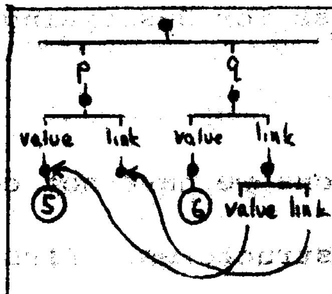
```
C = COMPLEX(1,2)                SNOBOL
A = R(C)
R(C) = 3

c ← [ r:1; i:2 ];               ML-3
a ← r of c;
r of c ← 3
```

Fig. 3.6-17. Example of data structures in SNOBOL.

But semantic rules which would elaborate on such properties are not to be found; instead, all that can be seen are a few examples. As with ISWIM, careful examination of the examples is required to produce a consistent and unambiguous ML-3 representation for the data structuring facilities of SNOBOL4. Some detective work is needed here as well: each of the two books [Gris 71, Gris 73] provides insufficient information to make such a determination, but using both together, enough clues can be gathered to resolve possible ambiguities. An example is shown in figure 3.6-18.

The translation into ML-3 may be straightforward, but a number of other possible translations which would result in different sharing properties were ruled out only after painstaking examination of the examples in both books. Surely a discussion of sharing in these books could have shed much-needed light on the semantics of data structures in SNOBOL4.



SNOBOL

```
DATA('NODE(VALUE,LINK)')
P = NODE(5,)
Q = NODE(6,)
LINK(Q) = P
```

ML-3

```
p ← [ value:5; link:nil ];
q ← [ value:6; link:nil ];
link of q ← p
```

Fig. 3.6-18. Sharing properties in SNOBOL

<u>Completeness</u>

In this chapter, we defined a series of mini-languages and used them to model data structuring facilities in three representative programming languages. An important question to ask is how complete our modeling is. In other words, how thoroughly have we covered the approaches to data structures found in these three languages? At first glance, our treatment seems rather incomplete because of the limited expressive power of the mini-languages we defined. But most of the features not included in our mini-languages are independent of the notions of data structures in the sense that the way such features are defined in an actual programming language has no bearing on how the language approaches concepts of data structures. The fact that our mini-languages lack character strings and conditional expressions, for instance, does not reflect on their completeness for describing data structures.

In PAL, there are only two notions we have not covered which have a direct bearing on data structures. First, arbitrary integer-valued expressions can be used to select components from a tuple. For example, the selection x n refers to the component of the tuple x whose selector is the value of the variable n. This cannot be translated into our mini-languages, which allow only constant ⟨selector⟩s (the ML-3 ⟨selection⟩ n of x would look for a component with selector "n"). The second uncovered feature in PAL is the built-in function Order, which when applied to a tuple yields the number of components in the tuple.

Neither of these two notions can be expressed in our mini-languages, but it was not our goal to be able to do so. For these two data structuring features, the semantic issues are well understood; we don't really need to treat them in our mini-languages. Extending the mini-languages to handle extra notions like these would only serve to ruin the syntactic and semantic simplicity of the mini-language approach.

In QUEST, the only data-structuring features we did not treat are the use of expressions to select components from a list, and several built-in functions that operate on lists. As with PAL, we feel that the issues raised here are outside the area of our main concern.

With SNOBOL4, we completely neglected the area of arrays. Although arrays are highly relevant to the issues we are interested in, they present some difficult problems for whose solutions additional mechanisms are needed. We discuss some of these problems in Chapter 5.

The three languages covered in this section are all "typeless" languages in the sense that there are no declarations associating identifiers with particular data types. In the next chapter, we deal with "typed" languages and some new semantic issues they introduce.

Chapter 4

DATA TYPES AND TYPECHECKING

4.1. Why We Want a Type System

In this chapter we will add a new facet to the design of our previous mini-languages. Consider the ML-3 ⟨assignment⟩ y ← x, which directs that the contents of the cell for x be placed into the cell for y. We translated this ⟨assignment⟩ into an invocation of the BL procedure assign3 (defined back in fig. 3.5-1). Every time this procedure is called, there is a separate set of tests performed to check whether the cell for the first parameter (which corresponds to x) contains an integer or a structure. The set of BL instructions chosen to perform the assignment operation depends on the result of these tests. In practice, however, a programmer will usually know in advance whether the identifier x will take on integer or structured values. This knowledge makes these runtime type tests in assign3 superfluous. We would like some way of telling the translator not to make such tests where they are not needed.

The technique of <u>static typechecking</u> achieves these goals. Its basic idea is to partition the set of values into convenient subsets called types. The translator can be informed of the programmer's intentions of keeping values only of a certain type in some given cell. With this knowledge, redundant runtime type tests can be eliminated. But it is still necessary to prevent type errors. For example, suppose we tell the translator that the variable x will take on only structured values. Each time we access the value of x, the BL code produced by the translator will fetch the components of x. If we somehow place an integer value in the cell bound to x, then during execution the interpreter would attempt to extract components where there are none, yielding undefined, probably erroneous results. To prevent such type errors from occurring, we would like to have the translator test each ⟨assignment⟩ to make sure it couldn't specify the placing of a value of one type into a cell intended to hold values of another type. Any ⟨program⟩ containing ⟨assignment⟩s which fail this test is invalid; the translator will notify the user of such an error in the same way that it flags syntactically erroneous ⟨program⟩s.

In testing ⟨assignment⟩s for validity, it will be useful for the translator to know for each ⟨destination⟩ the type of values intended to be stored in the associated cell.

This criterion can help us decide how to partition the ML-3 values into types. If we divide values into just two types, integers and structures, then the above criterion is not always satisfied. Suppose the ⟨identifier⟩ x is specified as assuming only structured values. Then the values yielded by both of the ⟨expression⟩s [a:3; b:4] and [a:3; b:[c:5; d:6]] can be stored in the cell bound to x, but we cannot say anything about the type of the ⟨destination⟩ b <u>of</u> x. In one case it has an integer value; in the other case, a structure. Thus finer type classifications are called for. We will want to ascertain from the type of a structured value what components it has and the type of each component. Such a type system is the basis for our next mini-language.

4.2. Mini-Language 4 -- Static Typechecking

Mini-Language 4 (ML-4) adds the notions of data types and static typechecking to the concepts we developed in the previous chapter. Specifically, it is an extension to ML-3, associating to every ⟨expression⟩ and to every cell a particular data type. For our purposes, we consider data types as sets of values. The set of integers is an ML-4 data type. Further, the set of all structured values with a given set of component ⟨selector⟩s such that the type of the component associated to each specific ⟨selector⟩ is given also is an ML-4 type. With this collection of data types, if we associate a type to each ⟨identifier⟩ mentioned in a ⟨program⟩, then we shall be able to determine the type associated with each cell referred to in the ⟨program⟩. Moreover, for any particular data type, one can determine whether the value yielded by a given ⟨expression⟩ belongs to this type.

Syntax of ML-4

The rules here govern the syntax of that part of ML-4 which is not found in ML-3 (namely the type system). We introduce the new primitive syntactic class ⟨typename⟩ to denote the set of underlined alphanumeric strings beginning with a letter. The distinguished ⟨typename⟩ int has particular significance, which will be discussed below.

```
⟨program⟩    ::= ⟨prelude⟩ ; ⟨assignment⟩ ;...; ⟨assignment⟩
⟨prelude⟩    ::= ⟨defn⟩ ;...; ⟨defn⟩ ; ⟨decl⟩ ;...; ⟨decl⟩
⟨defn⟩       ::= ⟨typename⟩ = ⟨structype⟩
⟨structype⟩ ::= [ ⟨comp decl⟩ ;...; ⟨comp decl⟩ ]
⟨comp decl⟩ ::= ⟨typename⟩ ⟨selector⟩
⟨decl⟩       ::= ⟨typename⟩ ⟨identifier⟩ ;...; ⟨identifier⟩
```

The remainder of the ML-4 syntax is identical to the syntax presented for ML-3, with two exceptions. First, ML-4 has no

⟨modification⟩s (which we simply won't have occasion to make use of), and second, ⟨construction⟩s appear slightly different:

⟨construction⟩ ::= ⟨typename⟩ [⟨field⟩ ;...; ⟨field⟩]
⟨field⟩ ::= ⟨cell expr⟩

(The ⟨selector⟩s that no longer expliqitly appear in the ⟨field⟩s of a ⟨construction⟩ may be found in the ⟨defn⟩ for the ⟨typename⟩ of the ⟨construction⟩.)

Description

We need to interpret the new syntactic classes. A ⟨program⟩ in ML-4 is essentially a ⟨program⟩ in ML-3, preceded by a ⟨prelude⟩. The ⟨prelude⟩ is a sequence of type definitions (⟨defn⟩s) followed by a sequence of declarations (⟨decl⟩s). A ⟨decl⟩, consisting of a ⟨typename⟩ and a list of ⟨identifier⟩s, specifies that those ⟨identifier⟩s are to assume values only of the type given by the ⟨typename⟩. Types in ML-4 are denoted by members of two syntactic classes as follows:

(1) A ⟨typename⟩ is either the symbol int (which denotes the type consisting of integer values) or the name associated with some type by a ⟨defn⟩.

(2) A ⟨structype⟩ denotes a structured type (i.e. a type consisting of structured values). The ⟨selector⟩s and types of the associated components of a value of such a type are specified by the

<comp decl>s (component declarations) in a <structype>.

Observe that if we know the type of a structured value, then we know the type of each of its components. There are two basic purposes for using <typename>s: first, to provide for multilevel structures (i.e. structures with components which are structures), and second, to allow for recursion in type definitions. We discuss recursive types later.

Semantics of ML-4 (informal)

(1) Data types and type definitions: We define the data types that are specified by the syntactic units of ML-4. Elements of the classes <typename> and <structype> define data types according to three rules:

(i) The <typename> int denotes the class of all integer values.

(ii) Suppose $s_1,\ldots,s_k$ are <selector>s and $t_1,\ldots,t_k$ are syntactic items denoting data types. Then the <structype> $[t_1 s_1;\ldots;t_k s_k]$ denotes the class of all structures with exactly k components with <selector>s $s_1,\ldots,s_k$ such that for each $i = 1,\ldots,k$ the value (if any) contained in the component cell selected by s_i belongs to the type t_i.

(iii) If t is the <typename> of a <defn>, then t denotes the type specified by the <structype> of that <defn>. In this case we say that the <defn> defines the <typename> t.

These rules give the semantics for type definitions in ML-4.

Note that according to rule (ii), if x is a value belonging to a structured type t, then the types of all the component cells of x are determined.

As examples, the objects of figure 4.2-1 belong to the type int. In the presence of the ⟨defn⟩s pt = [int p] and t = [int a; pt b], the objects depicted in figure 4.2-2 belong to the type t (which is the class of all two-component structures with a-component of type int and with b-component a one-component structure whose p-component is of type int). Note particularly that a cell constrained by our type mechanism to hold values of a given type can be empty. A value may belong to more than one type (particularly if it is a structure some of whose component cells are empty). But given any value v and any type t, one can always tell whether or not v belongs to t.

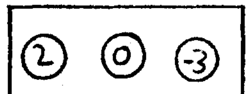


Fig. 4.2-1. Objects of type int

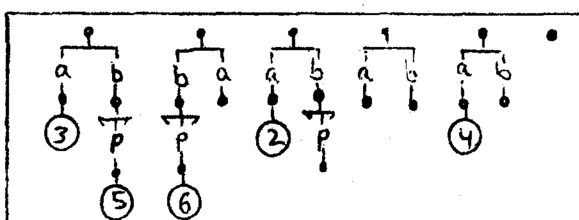


Fig. 4.2-2. Six objects of type t = [int a; pt b] (where pt = [int p]).

A ⟨typename⟩ does not have to be defined textually before it is used in a ⟨prelude⟩. For instance, the ⟨defn⟩ sequence <u>t1</u> = [<u>t2</u> c]; <u>t2</u> = [<u>int</u> d; <u>int</u> e] is perfectly legal. A nontrivial application is the definition of recursive data types, which arise in ML-4 when a ⟨typename⟩ is used as part of the ⟨structype⟩ in its definition. Consider, for example, the ⟨defn⟩ <u>r</u> = [<u>int</u> a; <u>r</u> b]. This defines a type named <u>r</u> consisting of two-component structures for which the a-component cell can hold only integer values and the b-component cell can hold values only of type <u>r</u>. Although it sounds circular, it is perfectly well defined. Values of a recursively defined type can have substructures nested to an arbitrary depth, and BL-objects representing such values frequently contain directed cycles.

We make three restrictions on ⟨defn⟩s in ML-4. First, the ⟨selector⟩s occurring in a ⟨structype⟩ must be distinct. Second, a ⟨typename⟩ can be defined only once in a ⟨program⟩. Third, the ⟨typename⟩ <u>int</u> must not be redefined. Any ⟨program⟩ not obeying these restrictions is syntactically invalid (i.e. is to be rejected by the translator). The meaning of an invalid ⟨program⟩ is undefined.

(2) <u>Declarations</u>: As with ⟨defn⟩s, the semantics for a ⟨decl⟩ does not specify any particular actions to be per-

formed at runtime. The effect of a ⟨decl⟩ is to cause the ⟨identifier⟩s in it to be associated with the type named in the ⟨decl⟩.

In order for a ⟨program⟩ to be syntactically valid, every ⟨identifier⟩ occurring in some ⟨assignment⟩ must appear exactly once in the ⟨program⟩'s ⟨decl⟩s. Every ⟨typename⟩ occurring in some ⟨decl⟩ must be defined exactly once in the ⟨defn⟩s.

From the above semantic rules for ⟨defn⟩s and ⟨decl⟩s, it is possible to uniquely determine the type of any ⟨expression⟩ in a syntactically valid ⟨program⟩. This is done as follows:

(i) Suppose the ⟨expression⟩ is a ⟨destination⟩. If it is an ⟨identifier⟩, then this ⟨identifier⟩ occurs in exactly one ⟨decl⟩ and its type is given by the ⟨typename⟩ of the ⟨decl⟩. If it is a ⟨selection⟩, then it consists of a ⟨selector⟩ and an ⟨expression⟩. The type of the ⟨expression⟩, which can be determined recursively, will be some structured type designated by a ⟨structype⟩. The type of the ⟨selection⟩, then, is given by the ⟨typename⟩ in the ⟨comp decl⟩ of the ⟨structype⟩ that contains the given ⟨selector⟩.

(ii) If the ⟨expression⟩ is a ⟨generator⟩, there are two cases: ⟨integer⟩s are of type int and ⟨construction⟩s are of the type given by their ⟨typename⟩.

Thus we can determine from the ⟨prelude⟩ of a syntactically valid ⟨program⟩ the type of any ⟨expression⟩; this type is given by precisely one ⟨typename⟩. For example, in the presence of the ⟨prelude⟩ xtype = [int a; ytype b]; ytype = [int c; int d]; xtype x; ytype y the type correspondences shown in figure 4.2-3 are valid.

Expression	Type
x	xtype
a of x	int
b of x	ytype
c of y	int
d of b of x	int
3	int
ytype[3;4]	ytype
xtype[5; ytype[6;nil]]	xtype

Fig. 4.2-3. Types of sample ⟨expression⟩.

(3) Assignments: the semantics of an ML-4 ⟨assignment⟩ specifies the same runtime actions as its ML-3 counterpart; in addition, the translator is directed to perform certain additional tests. An ⟨assignment⟩, as before, consists of a ⟨destination⟩ and an ⟨expression⟩. The ML-4 type system forces the cell referred to by the ⟨destination⟩ to hold values only of a certain type. Thus the translator must verify that the value of this ⟨expression⟩ matches this type.

A ⟨construction⟩ in which the components fail to match the types of the corresponding fields in the ⟨defn⟩ of its ⟨typename⟩ is an invalid ⟨expression⟩ and has undefined type. For example, if we define z = [int a; int b], then the ⟨construction⟩ z[1;2;3] is invalid because of its extra component; the ⟨construction⟩ z[1; z[2;3]] is also invalid because its b-component is of type z rather than int as required. We also call a ⟨construction⟩ invalid if its ⟨typename⟩ is not defined in the ⟨prelude⟩.

An ML-4 ⟨program⟩ is invalid if in any of its ⟨assignment⟩s the type of the ⟨expression⟩ is undefined or fails to match the type of the ⟨destination⟩. Each of these two types is given by precisely one ⟨typename⟩; these types are defined to match if and only if their ⟨typename⟩s are identical. The mechanisms we shall define for the translator insure that it can always determine whether or not a given ML-4 ⟨program⟩ is valid. There is no need for runtime type tests, nor are there any runtime type errors. However, a runtime error will occur if there is an attempt to extract components from an empty cell of a structured type. For instance, the ML-4 ⟨program⟩ s1 = [int a; s2 b]; s2 = [int c]; s1 x; x ← s1[3;nil]; c of b of x ← 4 will fail on interpretation of its last ⟨assignment⟩ (since the interpreter will look for a nonexistent c-component in the empty cell for b of x) even though the type of the ⟨destination⟩ c of b of x (int) matches the type of the ⟨expression⟩ 4. Thus we require runtime tests to check the ⟨selection⟩s in ML-4. Generally speaking, testing for empty cells is usually much easier than testing the type of the contents of a cell at runtime.

If we strip off the ⟨prelude⟩ from a valid ML-4

<program>, then we will have in essence an ML-3 <program> in which each cell takes on values of only one type. Moreover, the effect of executing this ML-4 <program> is identical to the effect of executing its ML-3 equivalent.

Translation into BL

To give a precise formulation for the semantics of ML-4, we describe the translation of ML-4 <program>s into BL. With the previous mini-languages, it sufficed to show the BL code corresponding to various program constructs, namely the different kinds of assignment statements. This is no longer sufficient in the case of ML-4, since the semantics now contains rules for typechecking by the translator. We must therefore also describe the typechecking procedures performed by the ML-4 translator.

In discussing how the translator performs typechecking of ML-4 <program>s to determine their validity, we begin by describing the information supplied to the translator by the <prelude> of a <program>. We shall treat the translator as a BL procedure. As it processes the <prelude>, the translator builds two component objects in its local structure: one component named $defns which represents the type definitions, and one named $decls which corresponds to the

the declarations. $defns is a structure which has one component for each ⟨typename⟩ found in the ⟨prelude⟩. Each component of $defns is a structure with information on the type associated with the ⟨typename⟩. For each ⟨typename⟩ defined in a ⟨defn⟩, the corresponding component of $defns has an "n" field with the number of components in a value of that type, numbered fields giving the ⟨selector⟩s of the components in the proper order, and a "val" field giving the types of the components (by means of links to the proper entries in $defns). The <u>int</u>-component of $defns has only a val-component containing the elementary value 'int'. $decls is a structure with one component for each ⟨identifier⟩ declared in the ⟨prelude⟩. If, say, the ⟨identifier⟩ x is declared to have type t, then the x-component of $decls shares with the t-component of $defns. In each of figures 4.2-4, 4.2-5 and 4.2-6 we give a ⟨prelude⟩ and exhibit the objects $defns and $decls constructed by the translator from the ⟨prelude⟩. The type with ⟨typename⟩ <u>r</u> in figure 4.2-5 is recursively defined; observe that $defns has a directed cycle in this case.

Once these objects have been constructed by the translator, all the information required for typechecking is available. Each type to be associated with some cell referred to in the ⟨program⟩ is represented by a component node of $defns. Two types match iff they have the same

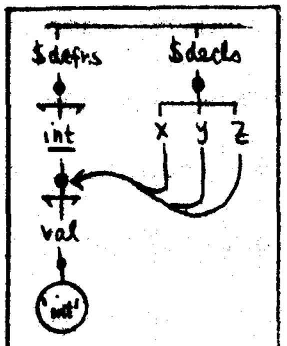


Fig. 4.2-4. ⟨prelude⟩ int x,y,z

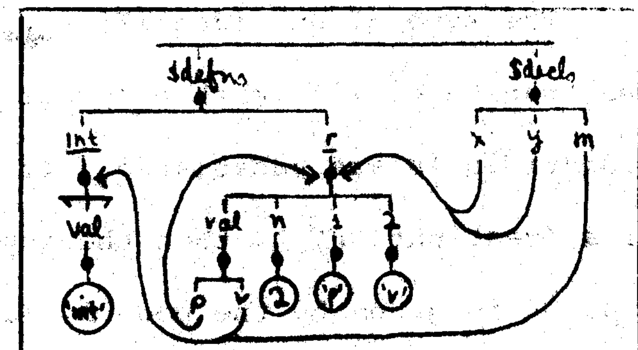


Fig. 4.2-5. $defns and $decls structures for the ⟨prelude⟩ r = [r p; int v]; r x,y; int m

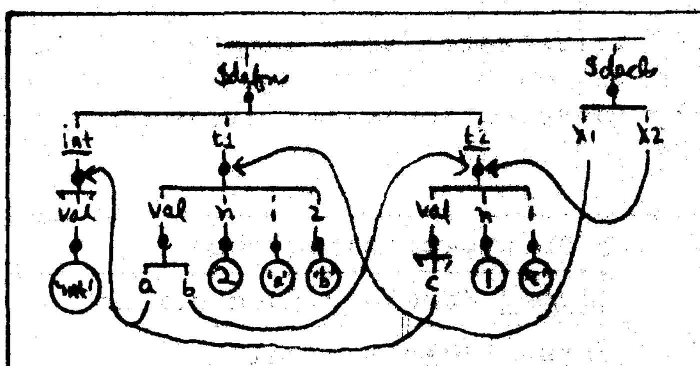


Fig. 4.2-6. $defns and $decls for the ⟨prelude⟩ t1 = [int a; t2 b]; t2 = [int c]; t1 x1; t2 x2

⟨typename⟩. To describe how the translator performs the actual typechecking, all that needs to be shown is how to access the node for the type of any ML-4 ⟨expression⟩; once we can do this, the typechecking is straightforward: an ⟨assignment⟩ has a type error iff the nodes for the types of its ⟨destination⟩ and its ⟨expression⟩ are distinct.

The type of an ⟨identifier⟩ x is given by $decls.x. The translator will mark a ⟨program⟩ invalid if any of its ⟨identifier⟩s are undeclared. If β is the node for the type of a ⟨destination⟩ D, then the type of the ⟨selection⟩ s of D is given by the node β.val.s. The translator verifies as part of its typechecking that values of the type of D do indeed have s-components. Thus we can ascertain the node for the type of any ⟨destination⟩ in an ML-4 ⟨program⟩. Figure 4.2-7 illustrates some sample ML-4 ⟨assignment⟩s involving only ⟨destination⟩s and gives BL typechecking code

ML-4 code	BL typechecking code
y ← x	same? $decls.y,$decls.x,no
z ← a of x	has? $decls.x.val,a,no same? $decls.z,$decls.x.val.a,no
b of y ← z	has? $decls.y.val,b,no same? $decls.y.val.b,$decls.z,no
b of y ← c of a of x	has? $decls.y.val,b,no has? $decls.x.val,a,no has? $decls.x.val.a.val,c,no same? $decls.y.val.b,$decls.x.val.a.val.c,no

Fig. 4.2-7. Examples of BL typechecking.

to determine their validity. A branch to the label "no" indicates that the ⟨assignment⟩ has a type error.

If an ⟨expression⟩ is an ⟨integer⟩, then its type is given by the node $defns.int. The type of a ⟨construction⟩ whose ⟨typename⟩ is t is given by the node $defns.t, provided the ⟨construction⟩ is valid. To check this, the types of the components in the ⟨construction⟩ must match the ⟨typename⟩s in the ⟨structype⟩ that defines t; moreover, there must be the same number of components in both places. Thus the translator can access by our scheme the node for the type of any ⟨generator⟩. As a result, we now see how the translator accesses the nodes for the types of arbitrary ML-4 ⟨expression⟩s. Figure 4.2-8 gives some examples of ML-4 ⟨assignment⟩s containing arbitrary kinds of ⟨expression⟩s; along with each ⟨assignment⟩ we show BL code which tests its validity. This completes our picture of how the translator performs static typechecking; the mechanisms should be clear from the examples in figures 4.2-7 and 4.2-8.

The actual BL code generated by the translator (i.e. the BL code to be interpreted at runtime during the execution of an ML-4 ⟨program⟩) is similar to what we presented in the section on ML-3. There are two differences reflect-

ML-4 code	BL typechecking code
x ← 2	same? $decls.x,$defns.int,no
z ← t[2]	same? $decls.z,$defns.t,no const 1,$temp /* values of type t must have exactly one component */ eq? $defns.t.n,$temp,no select $defns.t,1,$temp /* name of 1st component ⟨selector⟩, type t */ same? $defns.t.val.*$temp,$defns.int,no
w ← r[share w; x]	same? $decls.w,$defns.r,no const 2,$temp eq? $defns.r.n,$temp,no select $defns.r,1,$temp same? $defns.r.val.*$temp,$decls.w,no select $defns.r,2,$temp same? $defns.r.val.*$temp,$decls.x,no
y ← s[t[b of w]]	same? $decls.y,$defns.s,no const 1,$temp eq? $defns.s.n,$temp,no select $defns.s,1,$temp same? $defns.s.val.*$temp,$defns.t,no const 1,$temp eq? $defns.t.n,$temp,no has? $decls.w.val,b,no select $defns.t,1,$temp same? $defns.t.val.*$temp, $decls.w.val.b,no

Fig. 4.2-8. More examples of BL typechecking

ing the switch of typechecking from runtime to translate-time. First, occurrences of ⟨selection⟩s in ML-3 yield run-time type tests, such as the BL code has? x,b,error for the ML-3 ⟨selection⟩ b of x. In ML-4 this runtime type test is replaced by the simpler and faster test nonempty? x,error, which makes sure there is no erroneous attempt to access component cells of an empty cell.

The second change is that the complicated procedure assign3 with all its type tests is not needed at all. The BL code generated from the ⟨assignment⟩ y ← x depends on the type of the ⟨destination⟩ y. If its type is int, then by virtue of the translator's static typechecking we know that x can hold only integer values. In this case the BL code in figure 4.2-9 is generated. If y is of a structured type, then the translator knows that its ⟨selector⟩s $s_1, \ldots, s_k$ are given by

```
        clear       y
        nonempty?   x,skip
        const       *x,y
skip: ...
```

Fig. 4.2-9. BL code for the ML-4 ⟨assignment⟩ y ← x when y is int

s_1 = *($decls.y.1) , ... , s_k = *($decls.y.*($decls.y.n)).

In this case the BL code in figure 4.2-10 is generated. The translator can always tell which case applies by testing whether the pathnames $decls.y and $defns.int lead to the same cell. The BL instruction same? $decls.y,$defns.int,go performs this test. A branch to the label "go" indicates that y has a structured value and that the second case applies. Thus, by substituting the nonempty? test for the has? test and the BL code of figures 4.2-9 and 4.2-10 for invocations of the assign3 procedure, we obtain the BL code yielded by the ML-4 translator. This completes our description of the translation of ML-4 into BL and places the semantics of ML-4 on firm and precise ground.

```
        clear      y
        nonempty?  x,skip
        link       y,*s1,x.*s1
           :
        link       y,*sk,x.*sk
skip: ...
```

Fig. 4.2-10. BL code for the ML-4 ⟨assignment⟩ y ← x when y is structured

4.3. Discussion and Examples

Most programming languages handling data structures have a type system similar to that of ML-4; the bulk of their typechecking is done at translation time rather than runtime. In this section we treat the data structuring facilities of three of these languages, using ML-4 as a vehicle for describing their semantics.

ALGOL W

The language ALGOL W [Wir 66] has a relatively simple treatment of data structures. The structures are called *records*, and the ALGOL W analog to an ML-4 structured type is called a *record class*. An ALGOL W record class declaration can be represented by an ML-4 ⟨defn⟩. Figure 4.3-1 shows how the two languages define classes of structured objects; the ML-4 type with ⟨typename⟩ *pair* corresponds to the ALGOL W record class named pair. Structured objects are built in ALGOL W through the use of *record designators*, which are analogous to ML-4 ⟨construction⟩s. Expressions in both languages which build structures from the "pair" class are also shown in figure 4.3-1.

language	type definition	object construction
ALGOL W	*record* pair (*integer* a,b)	pair (3,4)
ML-4	*pair* = [*int* a; *int* b]	*pair*[3;4]

Fig. 4.3-1. A parallel between ALGOL W and ML-4.

There is a major difference between ALGOL W and ML-4 with respect to these elements. Although a record designator builds a structured object in ALGOL W, it does not yield as its value the object it constructs. In fact, records are not even values in ALGOL W. A record class is not a legitimate type in ALGOL W; records are accessed through values of *reference* types. For instance, the ALGOL W record designator pair(3,4) in figure 4.3-1 yields a value of type <u>reference</u>(pair). ML-4 will treat reference expressions in ALGOL W similarly to the way ML-3 treats pointers in ML-2. The correspondence is depicted in figure 4.3-2. Note that in dealing with ALGOL W records, we need an extra level of indirection (the "ptr" component). This (at least with respect to our scheme of representation) is the same kind of inefficiency we encountered with ML-2. It is worse here, though, since ML-2 made use of the indirection only when sharing was needed.

ALGOL W

```
record pair (integer a,b);
reference(pair) y,z;
y := pair(3,4); z := y
```

ML-4

```
pair = [ int a; int b ];
refpair = [ pair ptr ];
refpair y,z;
y ← refpair[ pair[3;4] ];
z ← y
```

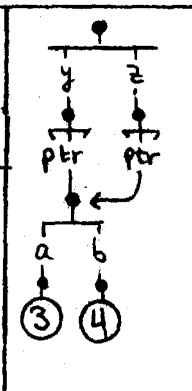


Fig. 4.3-2. Reference expressions in ALGOL W.

Components of a record can be accessed by selector functions in ALGOL W. Figure 4.3-3 shows the correspondence between selections in ALGOL W and ML-4 (z is of type <u>reference</u>(pair) in ALGOL W, <u>refpair</u> in ML-4).

language	selection
ALGOL W	a(z)
ML-4	a <u>of</u> ptr <u>of</u> z

Fig. 4.3-3. Selection.

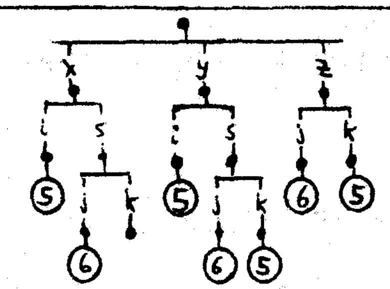
Once these differences concerning the construction and selection operations have been taken into account, we find that assignment, sharing and typechecking in ALGOL W are almost identical to the "obvious" ML-4 counterparts (e.g. replace ":=" with "←"). In this respect, ALGOL W is similar to the language SNOBOL4 described in section 3.6.

PL/1

PL/1 was one of the earliest languages to have compile-time typechecking and to treat both data structures and pointers. Most PL/1 constructs handling these notions look markedly different from the constructs we have seen in

PL/1

```
DECLARE 1 X,
          2 I FIXED BIN,
          2 S,
           3 J FIXED BIN,
           3 K FIXED BIN;
DECLARE Y LIKE X;
DECLARE Z LIKE X.S;
X.I = 5; X.S.J = 6;
Y = X;
Y.S.K = X.I;
Z = Y.S;
```



ML-4

```
trip = [int i; pair s];
pair = [int j; int k];
trip x,y;  pair z;
x ← trip[nil; pair[nil;nil]];
y ← trip[nil; pair[nil;nil]];
z ← pair[nil;nil];
i of x ← 5; j of s of x ← 6;
i of y ← i of x;
 j of s of y ← j of s of x;
 k of s of y ← k of s of x;
k of s of y ← i of x;
j of z ← j of s of y;
k of z ← k of s of y
```

Fig. 4.3-4. Structures in PL/1.

other languages. Figure 4.3-4 shows how PL/1 handles a sample structure and gives an ML-4 equivalent. We make two observations. First, all component cells of the PL/1 structures in this example are allocated when the declarations are interpreted. With ML-4, component cells are allocated when the structured value is actually constructed. Second, a PL/1 structure assignment like Y = X in fig. 4.3-4 signifies component-by-component copying (recursively for structured components) as with ML-1 and [illegible].

Unlike ALGOL W, there is no sharing among PL/1 structures until we introduce pointers and the attribute BASED. If P is a PL/1 variable declared to be a pointer, then declaring a structured variable with the attribute BASED(P) introduces a vast conceptual difference. This variable no longer signifies a location where structured objects may be stored; instead, it plays the role of a structured type. Figure 4.3-5 exhibits a set of PL/1 declarations involving BASED structures and gives a corresponding ML-4 (prelude) and set of ALGOL W declarations.

Although the PL/1 declarations of figure 4.3-4 specify allocation of storage to hold structured values (and allocation of component cells as well), the declaration of LIST in figure 4.3-5 does no such thing. BASED structure values in PL/1 are constructed through the use of an ALLOCATE statement. Under the declarations in figure 4.3-5, the PL/1 statement ALLOCATE LIST may be represented in ML-4 by the ⟨assignment⟩ p ← ptrlist[list[nil;nil;nil]]. Since LIST is declared to be BASED on the pointer P, the allocation causes the value of P to be set to point to the newly-built structure. The result of this allocation is shown in fig. 4.3-6.

```
PL/1
 DECLARE (P,H,T) POINTER;
 DECLARE 1 LIST BASED(P),
          2 BACK POINTER,
          2 FWD  POINTER,
          2 NUM FIXED BIN;

ML-4
 ptrlist = [list ptr];
 list = [ptrlist back;
         ptrlist fwd;
         int num];
 ptrlist p,h,t

ALGOL W
 record list =
   (reference(list) back;
    reference(list) fwd;
    integer num);
 reference(list) p,h,t;
```

Fig. 4.3-5. PL/1 BASED structures as types.

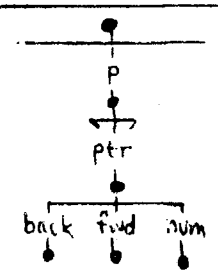


Fig. 4.3-6. Value of p.

BASED structures in PL/1 are accessed through pointers. In our LIST example, a use of the name LIST refers to whatever the pointer P is currently pointing to (which will be the most recently constructed structure BASED on P, unless P has been subsequently updated). To refer to a previous allocation, one must use a qualified reference such as T -> LIST (which indicates whatever the pointer T is currently pointing to). Figure 4.3-7 draws the connection between PL/1, ALGOL W and ML-4 in accessing fields of structures (it is assumed that the declarations in fig. 4.3-5 are still in force).

PL/1	ALGOL W	ML-4
LIST	p	ptr of p
T -> LIST	t	ptr of t
LIST.NUM	p.num	num of ptr of p
T -> LIST.NUM	t.num	num of ptr of t

Fig. 4.3-7. Accessing fields.

The meaning of assignment in PL/1 is similar to ALGOL W except for its handling of structured values (which ALGOL W does not choose to handle). In this case, as we have said, PL/1 copies rather than induce sharing. All sharing of data in PL/1 is done through pointers.

Typechecking in PL/1 differs from ML-4 and ALGOL W in one major area, that of pointers. The ALGOL W translator insures that a reference value can point to records only from one record class; if c1 and c2 are distinct record classes, then any attempt to make a value of type reference(cl) point to a record from class c2 will be caught by the translator and marked as illegal. The type system for ML-4 imposes essentially the same restrictions. However, a variable of type POINTER in PL/1 can be set to point to values of any type at any time (including nonstructured values). This causes difficulties of the same kind that static typechecking is supposed to eliminate. For example, in the PL/1 program segment of figure 4.3-8, the assignment P = Q is legal, even though P points to a structure of type

PL/1	ML-4
DECLARE (P,Q) POINTER;	
DECLARE 1 M1 BASED(P),	ml = [int j; int k];
2 J FIXED BIN,	ptrml = [ml ptr];
2 K FIXED BIN;	ptrm2 = [int ptr];
DECLARE M2 FIXED BIN BASED(Q);	ptrml p; ptrm2 q;
ALLOCATE M1;	p ← ptrml[ml[nil;nil]];
ALLOCATE M2;	q ← ptrm2[nil];
P = Q;	p ← q;
M1.K = 5;	k of ptr of p ← 5

Fig. 4.3-8. Lack of type restrictions on PL/1 pointers.

M1 and Q points to the integer M2. The reference to M1 in the following line (M1.K = 5) designates whatever P will be pointing to (which is the integer M2 since P has just been assigned the value of Q). Thus there will be (depending on the implementation) a runtime error or at least an erroneous result as an outcome of the attempt to update a component of the integer value M2. The ML-4 translation of this program, also shown in figure 4.3-8, is invalid since in the <assignment> p ← q the types fail to match (ptrm1 vs. ptrm2). If in the PL/1 program we had declared M2 to be BASED on P, then the corresponding ML-4 <program> would have two conflicting declarations for p, which would also render it invalid. Thus we see that the typechecking system in PL/1 fails to catch a whole class of programs which might have runtime type errors.

ALGOL 68

The treatment of data structures and pointers in ALGOL 68 is linked to an intricate system of types and typechecking. ALGOL 68 is a difficult language to learn and understand; the defining documentation [VWij 69; VWij 73] presents an intimidating formalism to the uninitiated. However, there are works (e.g. [Lind 71]) which are immensely helpful.

Types in ALGOL 68 are called modes. The modes of relevance to us are the mode int (integer values) and the modes built from the mode-constructors struct and ref (structured and reference values, respectively). We describe a correspondence which assigns ML-4 types to ALGOL 68 modes:

(1) To the ALGOL 68 mode int we assign the ML-4 type int.

(2) If $M_1,\ldots,M_k$ are modes and $S_1,\ldots,S_k$ are tags (the equivalent of ⟨selector⟩s), then to the mode struct(M_1 S_1,...,M_k S_k) we assign the ML-4 type [T_1 S_1;...;T_k S_k], where the T_i are the ML-4 types corresponding to the M_i.

(3) If M is a mode then to the mode ref M we assign the type [T ptr], where T is the ML-4 type corresponding to M.

Mode-declarations in ALGOL 68 are just like type definitions in ML-4; for example the mode-declaration mode pair = struct(int a, int b) is equivalent to the ML-4 ⟨defn⟩ pair = [int a; int b].

A declaration in ALGOL 68, besides associating an identifier with a mode and imposing type restrictions on the rest of the program, has a two-fold runtime effect. Consider a declaration of form M X = E, for instance int x = 3, where M is a mode, X an identifier, and E an expression yielding a value of mode M. This declaration first binds X to a newly-allocated cell. Second, it places the mode M value yielded by E into this cell. What is peculiar about ALGOL 68 declarations is that this value can never be changed. It may, however, be a reference value (i.e. the mode M is ref N for some other mode N); in this case it refers to (points to) a cell holding values of mode N. This latter cell (and not the former cell) can be updated by the assignment operation in ALGOL 68. Thus the meaning of assignment in ALGOL 68 differs from assignment in the other languages we have discussed. Note that an identifier whose declared mode is not a reference mode serves essentially as a constant. An identifier of mode <u>ref</u> N in ALGOL 68 plays the same role as a variable of type N in another programming language.

The specific definition of ALGOL 68 assignment is as follows: let E be an expression yielding a value of mode M (M can be arbitrary) and D an expression of mode <u>ref</u> M. The value of D is a reference to a cell which can hold values of mode M. Then D := E is a valid assignment and specifies that the mode-M value of E is to be stored in the mode-M cell referred to by (the value of) D.

A particular kind of ALGOL 68 expression, known as a <u>local generator</u>, specifies allocation of a new cell when it is evaluated. If M is a mode, then evaluation of the local generator <u>loc</u> M causes a new cell (which can only hold values of mode M) to be allocated. The value yielded by <u>loc</u> M is a reference to this new cell and therefore belongs to the mode <u>ref</u> M.

To obtain a variable in ALGOL 68 which will take on values of a mode M, we must declare an identifier X of mode ref M so that assignment can change the mode-M values. This may be accomplished by means of an ALGOL 68 declaration of form M X, which is really an abbreviation for the declaration ref M X = loc M. Consider, for example, the ALGOL 68 declaration int x (equivalent to the declaration ref int x = loc int), whose effect is depicted in figure 4.3-9. The identifier x, which is declared here to be of mode ref int, is bound to the upper cell; the lower cell is allocated (by evaluating loc int in ALGOL 68, and by evaluating the

ALGOL 68

```
{int x
{ref int x = loc int}
```

ML-4

```
refint = [int ptr];
refint x;
x ← refint[nil]
```

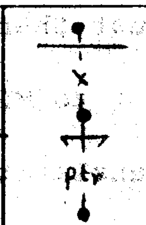


Fig. 4.3-9. Semantics of the ALGOL 68 declaration int x.

⟨cell expr⟩ nil in the ⟨construction⟩ refint[nil] in ML-4); and the upper cell receives as (permanent) value a pointer to the lower cell. Subsequent execution of the ALGOL 68 assignment x := 3 would place the value 3 in the lower cell; therefore its ML-4 equivalent is the ⟨assignment⟩ ptr of x ← 3. The static typechecking rules for ALGOL 68 insure that any assignment attempting to place a non-integer value in the lower cell is detected and indicated to be invalid.

There is one aspect of the ALGOL 68 type system which is more lenient than the ML-4 system. Unlike PL/1, no type errors can arise from this loosening. Consider the assignment y := x, where both identifiers x and y have been declared to be of mode ref int. This assignment specifies the updating of the mode int cell pointed to by y. But the right-hand side, which must then supply an integer value, is of mode ref int; according to ML-4 rules, the assignment is to be rejected by the translator as invalid. However, ALGOL 68 recognizes that the ref int value of x points to an int value, so all that needs to be done to obtain the required integer value is follow the pointer x. This process is called dereferencing. In general, the procedure for obtaining a value of a desired mode from a value of some other mode is known as coercion or conversion. Thus, in the ALGOL 68 type system, if the left-hand side of an assignment is of mode ref M, then the assignment is valid provided the right-hand side is of mode M or can be coerced to yield a mode M value. In our case, the procedure which translates from ALGOL 68 into ML-4 must recognize that dereferencing is called for, mark the assignment y := x as legal and generate ML-4 code which takes the coercion into account. Of the three assignments in the example shown in fig. 4.3-10, coercion takes place only in the second one (where y is dereferenced). The y on the right-hand side here is translated into the ML-4 ⟨expression⟩ ptr *of* y, yielding a valid ML-4 ⟨assignment⟩.

Note that the mode of x is *int*, and the mode of y and z is *ref* *int*.

```
ALGOL 68
  int x = 3;
  int y,z;
  y := x;
  z := y;
  y := 4;
ML-4
  refint = [int ptr];
  int x;
  refint y,z;
  x ← 3;
  y ← refint[nil];
  z ← refint[nil];
  ptr of y ← x;
  ptr of z ← ptr of y;
  ptr of y ← 4
```

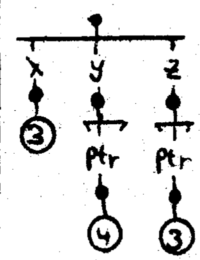


Fig. 4.3-10. An example of coercion in ALGOL 68.

The concept of structured values in ALGOL 68 is essentially the same concept when taken by itself as in ML-1 and ML-2 (as well as PL/1 and QUEST). Sharing arises only through the use of reference modes; assignment of structured values is done by componentwise copying. Figure 4.3-11 gives an example. The mode of z is *pair*; the mode of x is ref pair. The expression (5,6) in the declaration for z is called a structure display and simply gives values for the components of z.

ALGOL 68

```
mode pair = struct(int a,b);
pair z = (5,6);
pair x;
x := z;
```

ML-4

```
pair = [int a; int b];
refpair = [pair ptr];
pair z; refpair x;
z ← pair[5;6];
x ← refpair[pair[nil;nil]];
a of ptr of x ← a of z;
b of ptr of x ← b of z
```

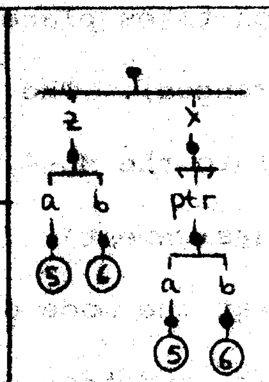


Fig. 4.3-11. Structure assignment in ALGOL 68.

The selection of components from a structure in ALGOL 68 is syntactically identical to ML-4. In fig. 4.3-11, the selection b of z, which refers to the b-component cell of z, is of mode int. There is a major complication concerning selection in ALGOL 68. We can legally form the selection b of x, where x is of reference-to-structure mode. The mode of the selection b of x is ref int, not int even though the b-component cell for the structure pointed to by x in figure 4.3-11 is of mode int. We say in this case that the pointer is distributed over the components (in ALGOL 68 terminology, x is "endowed with subnames"). Thus, for example, the assignment b of x := a of z is legal; in the ALGOL 68 program of fig. 4.3-11 it would place the value 5 into the b-component cell of the structure pointed to by x.

Unfortunately, the "obvious" translation into ML-4 fails. The ML-4 type refint, defined as [int ptr], corresponds to the mode ref int, but in fig. 4.3-11 there is no cell of this type to associate to the ⟨destination⟩ that corresponds to the ALGOL 68 selection b of x. Thus, in translating from ALGOL 68 into ML-4, such cells must be added to the picture (these cells will hold pointers to the individual components of the structure referred to by x). The corrected translation mechanism is shown in fig. 4.3-12;

ALGOL 68

```
mode pair = struct(int a,b);
pair x;
a of x := 3;
b of x := a of x;
```

ML-4

```
pair = [int a; int b];
refpair = [pair ptr];
refint = [int ptr];
subpair=[refint a;refint b];
refpair x; subpair x$sub;
x ← refpair[ pair[nil,nil] ];
x$sub ← subpair[refint[share a of ptr of x];
                refint[share b of ptr of x]];
ptr of a of x$sub ← 3;
ptr of b of x$sub ← ptr of a of x$sub
```

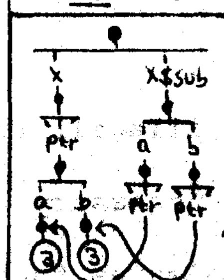


Fig. 4.3-12. Distributed pointers in ALGOL 68.

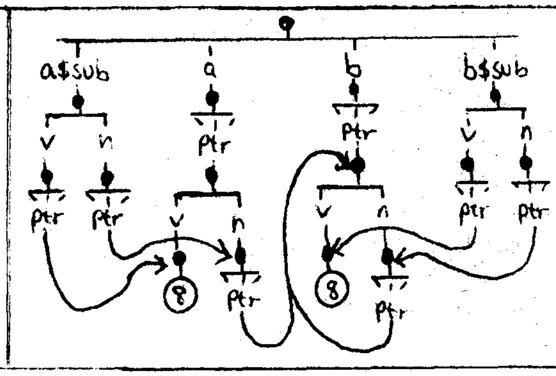
for each reference-to-structure identifier x we add to the local structure a reserved identifier x$sub to hold the subnames (distributed component pointers). By looking at the local structure pictured in fig. 4.3-12, we see that there are two ways to access component cells of the structure pointed to by x: through x (with ⟨destination⟩ b of ptr of x) as when updating the structure itself by componentwise copying; or through x$sub (with ⟨destination⟩ ptr of b of x$sub) as when explicitly selecting from x using subnames. Note that our translation conforms to the stipulations set by the ML-4 static typechecking system.

We give a final ALGOL 68 example, illustrating a recursive structured mode. The example is shown in figure 4.3-13. box is a structured mode, recursively defined, and a and b are of mode ref box. Note that the mode of the selection n of a is ref ref box. The only coercion in the program occurs in the last assignment, where a is dereferenced. A recursive mode definition such as mode badbox = struct(int v, badbox n) would be illegal; the "ref" inside the definition of the mode box is necessary since there is no implicit nil in ALGOL 68's modes as there is with ML-4.

Thus we see that even with a language as complex as ALGOL 68, we can use ML-4 to make clear its approaches to the semantics of data structures.

ALGOL 68

```
mode box = struct(int v,
                  ref box n);
box a,b;
v of a := 8;
n of a := b;
b := a;
```



ML-4

```
box = [int v; refbox n];  refbox = [box ptr];
subbox = [refint v; refrefbox n];
refint = [int ptr];  refrefbox = [refbox ptr];

refbox a,b;  subbox a$sub,b$sub;

a ← refbox[box[nil;nil]];  b ← refbox[box[nil;nil]];
a$sub ← subbox[refint[share v of ptr of a];
               refrefbox[share n of ptr of a]];
b$sub ← subbox[refint[share v of ptr of b];
               refrefbox[share n of ptr of b]];

ptr of v of a$sub ← 8;
ptr of n of a$sub ← b;
v of ptr of b ← v of ptr of a;
n of ptr of b ← n of ptr of a
```

Fig. 4.3-13. Final ALGOL 68 example.

Completeness

In this chapter, we defined the mini-language ML-4 and used it to model data structuring facilities of the languages ALGOL W, PL/1, and ALGOL 68. As in the last chapter, we close with a few remarks on the completeness of our coverage of the approaches to data structures found in these three languages.

With ALGOL W, as with SNOBOL4 in the previous chapter, we covered nearly all the data structuring facilities thoroughly, with the exception of arrays. We comment on arrays and some of their special issues in Chapter 5.

For PL/1 and ALGOL 68, our treatment is far from complete. This is to be expected because of the sheer bulk and complexity of these two languages. There are numerous features dealing with data structures which we have not described. Yet we claim that those features which we did describe in PL/1 and ALGOL 68 constitute the "heart" of their data structuring facilities; thus our description of these features should make clear the underlying semantic approaches to data structures in these languages as well.

Chapter 5

CONCLUSIONS AND EXTENSIONS

5.1. What We Have Done

There are a large number of programming languages which work with data structures. Because of the variety of approaches found in these languages, many subtle but important semantic distinctions crop up. With most languages, the semantics (including in particular the semantics for the data structuring facilities) are described informally in English. We consider such descriptive methods inadequate for our goals, since in many cases they fail to make clear some of the important semantic principles such as sharing. As we have seen, a misunderstanding of the interaction between notions such as assignment and sharing can lead the programmer into erroneous conclusions about the effects of programs.

We have therefore developed in this thesis a methodology for describing the semantics of data structures in programming languages. In order to precisely describe mechanisms found in programming languages which handle data structures, we made use of the base language model, which is an interpretive model for formal semantics. The base language model is essentially a mathematical formalism for modeling the changing states of a computing system on which various computations are performed. A mathematical treatment of the base language model is found in the Appendix; our approach emphasized the use of the base language as a programming tool similar to many conventional assembler languages. A major advantage of the base language model over other formal semantic models is that it manipulates data objects of a sufficiently general nature that we can make direct use of its data representations in our work without need for special encoding mechanisms.

The main portion of this thesis was concerned with the presentation and use of a series of mini-languages. With these mini-languages, we isolated the relevant conceptual abstractions such as assignment, value, construction, selection, sharing and typechecking. The mini-languages provided a "high-level" descriptive vehicle which made it simpler and more convenient to talk about semantic issues relating to data structures.

The basic structure of our methodology was to first make clear the semantics of our mini-languages by specifying their translation into the base language. Once this was done, we no longer needed to think in terms of the primitive operations of the base language. We were then able to describe the semantics of data structuring features in some programming language by simply using the appropriate mini-language to describe how the relevant mechanisms worked.

In treating the data structuring semantics of several programming languages, we gave mini-language code into which constructs of these languages are translated. Determination of this mini-language code presents difficulties when the semantics of the source language is incompletely or ambiguously specified, reflecting the inadequacy of the descriptive methods in use. Of course, once we have obtained a consistent translation into the right mini-language, we have an unambiguous semantic specification of the relevant constructs.

Using the techniques we developed, we described the data structuring semantics of a number of representative programming languages. With the simpler languages, we were able to give a nearly complete treatment of the data structuring facilities. As to the more complex languages, we were able to cover most of the fundamental approaches to data structures without getting caught up in the intricacies of features of relatively little semantic relevance to the issues we are concerned with. In the next section, we talk about some of the areas that were left uncovered.

5.2. Further Work

There are a number of semantic areas that we have not treated. In order to cover these areas, we would need to develop new mini-languages with additional mechanisms. In this section, we give brief mention to two such areas and what kinds of new mechanisms are required to treat them.

The first uncovered area is unions. With the type system of ML-4, every cell is constrained to hold values of only one type. In many programming languages, this restriction is weakened somewhat by defining union types. If type t is the union of types t1 and t2, then a cell of type t can hold values of type t1 as well as values of type t2. For example, suppose we declare z to be of type t in some language that admits union types, and suppose that the expressions e1 and e2 yield values of types t1 and t2, respectively. Then both the assignments z := e1 and z := e2 would be legal. This capability is not within the reach of the type mechanisms we developed for ML-4. Suppose we declare x to be of type t1. Then the assignment x := z can be executed without type error precisely when the value of z is of type t1 rather than of type t2. So in order to add to our mini-languages a capability to handle unions, some kind of additional runtime type testing mechanism must be introduced into the design of the language.

The second uncovered area is arrays. The type system of ML-4 is simply not equipped to deal with arrays whose subscript bounds are flexible. The type of such an array would contain structures having differing numbers of components. A structured type in ML-4 requires a set of selectors which is known to the translator and cannot change. Even with unions, we are no better off. For instance, the type consisting of all PAL tuples could not even be expressed as a finite union of ML-4 types, since a tuple can have any one of an infinite number of selector sets ({1}, {1,2}, {1,2,3}, ... , {1,2,...,n}, ...).

There are many other complicated issues concerning arrays, such as different array type concepts, changeability of bounds, and assignments between fixed and flexible arrays. All of these issues introduce new complexity into the language, requiring the development of more techniques.

To sum up, our methodology for describing data structures has special advantages from each of its two portions. The use of the base language model provides for a precise, formal characterization of the semantic rules of the languages under study, while our mini-languages provide the convenience of high-level descriptions of the actions being modeled. In order to describe any programming language feature, all that needs to be done is construct an appropriate mini-language which handles only the concepts directly relating to that feature. The syntax and semantics of such a mini-language are naturally easy to work with and understand. By specifying translations from source languages into the mini-language and from the mini-language into the base language, we gain a precise but conceptually clear characterization of the semantics of the features we wish to study.

Bibliography

[Amer 72] Amerasinghe, S.N. The Handling of Procedure Variables in a Base Language. S.M. thesis, M.I.T. Department of Electrical Engineering, Sept. 1972.

[Amer 73] ______. Translation of a Block Structured Language With Non-Local Go To Statements and Label Variables to the Base Language. M.I.T. Project MAC Computation Structures Group Memo 84, June 1973.

[Bur 68] Burstall, R.M. Semantics of Assignment. Machine Intelligence 2, ed. E. Dale and D. Michie. Oliver and Boyd, Edinburgh, 1968, 3-20.

[Cou 73] Coueignoux, P. and Janson, P. Translation of Simula 67 into the Common Base Language. M.I.T. Project MAC Computation Structures Group Memo 87, June 1973.

[Denn 71] Dennis, J.B. On the Design and Specification of a Common Base Language. M.I.T. Project MAC Computation Structures Group Memo 60, July 1971.

[Denn 74] ______. Private communication.

[Der 74] Dertouzos, M.L. Computer Languages: Structure and Interpretation. Class notes for subject 6.031, M.I.T. Department of Electrical Engineering, Feb. 1974.

[Dra 73] Drake, C. The Semantic Specification of SNOBOL in the Common Base Language. M.I.T. Computation Structures Group Memo 85, June 1973.

[Earl 71] Earley, J. Towards an Understanding of Data Structures. CACM 14, 10, Oct. 1971, 617-627.

[Ev 70] Evans, A. PAL Reference Manual and Primer. M.I.T. Department of Electrical Engineering, Feb. 1970.

[Fenn 73] Fenner, T.I. et. al. QUEST: The Design of a Very High Level Pedagogic Programming Language. ACM SIGPLAN Notices, Feb. 1973, 3-27.

[Floy 67] Floyd, R.W. Assigning Meanings to Programs. Proc. Symposium on Applied Mathematics. AMS, 1967, 19-32.

[Gris 71] Griswold, R.E. et. al. The SNOBOL4 Programming Language, 2nd. ed. Prentice-Hall, Englewood Cliffs, N.J., 1971.

[Gris 73] Griswold, R.E., and Griswold, M.T. SNOBOL4 Primer, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[Hoar 68] Hoare, C.A.R. Record Handling. Programming Languages, ed. F. Genuys. Academic Press, 1968.

[Hoar 69] ______. An Axiomatic Basis for Computer Programming. CACM 12, 10, Oct. 1969, 576-580,583.

[Hoar 71] ______. Proof of a Program: FIND. CACM 14, 1, Jan. 1971, 39-45.

[Hoar 72] ______. Notes on Data Structuring. Structured Programming, ed. E.W. Dijkstra. Academic Press, 1972.

[Lan 64] Landin, P. The Mechanical Evaluation of Expressions. Computer J., 6, 4, 1964, 308-320.

[Lan 65] ______. A Correspondence Between Algol 60 and Church's Lambda Notation. CACM 8, 2,3, Feb. and Mar. 1965, 89-101, 158-165.

[Lan 66a] ______. The Next 700 Programming Languages. CACM 9, 3, Mar. 1966, 157-166.

[Lan 66b] ______. A λ-Calculus Approach. Advances in Programming and Non-Numerical Computation. Pergamon Press, 1966.

[Lau 68] Lauer, P. Formal Definition of ALGOL 60. Technical report TR25.088, IBM Laboratory, Vienna, 1968.

[Lav 74] Laventhal, M. Verification of Programs Operating on Structured Data. M.I.T. Project MAC Technical Report TR-124, March 1974.

[Led 71] Ledgard, H.F. Ten Mini-Languages: A Study of Topical Issues in Programming Languages. ACM Computing Surveys 3, 3, Sept. 1971.

[Lee 72] Lee, J.A.N. Computer Semantics. Van Nostrand Reinhold, New York, 1972.

[Lind 71] Lindsey, C.W. and van der Meulen, S.G. Informal Introduction to ALGOL 68. Mathematisch Centrum, Amsterdam, 1971.

[Luc 68] Lucas, P., Lauer, P. and Stigleitner, H. Method and Notation for the Formal Definition of Programming Languages. Technical Report TR25.087, IBM Laboratory, Vienna, June 1968.

[Luc 69] Lucas, P. and Walk, K. On the Formal Description of PL/I. Annual Review of Automatic Programming 6, 3, 1969.

[McC 62] McCarthy, J. et. al. LISP 1.5 Programmer's Manual. The Computation Center and Research Laboratory of Electronics, M.I.T. M.I.T. Press, Cambridge, Mass., 1962.

[Morr 68] Morris, J.H., Jr. Lambda Calculus Models of Programming Languages. M.I.T. Project MAC Technical Report TR-57, 1968.

[Mos 74] Mosses, P. The Mathematical Semantics of Algol 60. Technical Monograph PRG-12, Oxford University Computation Lab, Programming Research Group, Jan. 1974.

[Reyn 72] Reynolds, J.C. Definitional Interpreters for Higher-Order Programming Languages. Proc. 25th ACM National Conference, 1972, 717-740.

[Reyn 73] ______. Towards a Theory of Type Structure, prelim. draft. Syracuse University, 1973.

[San 73] Sanderson, J.G. The Lambda Calculus, Lattice Theory and Reflexive Domains, draft notes, Oxford University Computing Laboratory, Programming Research Group, 1973.

[Scot 70] Scott, D. Outline of a Mathematical Theory of Computation. Proc. 4th Annual Princeton Conf. on Information Sciences and Systems, 1970, 169-176.

[Scot 71] ______, and Strachey, C. Towards a Mathematical Semantics for Computer Languages. Proc. Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, N.Y., 1971.

[Stra 66] Strachey, C. Towards a Formal Semantics. Formal Language Description Languages. North-Holland, 1966.

[Stra 67] ______. Fundamental Concepts in Programming Languages. NATO Conf., Copenhagen, 1967.

[VWij 69] Van Wijngaarden, A. (ed.) Report on the Algorithmic Language ALGOL 68. Numerische Mathematik, 14, 2, 1969, 79-218.

[VWij 73] ______. Proposed Revised Report on the Algorithmic Language ALGOL 68. IFIP, July 1973.

[Walk 69] Walk, K., et. al. Abstract Syntax and Interpretation of PL/I, Version III. Technical Report TR25.098, IBM Laboratory, Vienna, April 1969.

[Weg 68] Wegner, P. Programming Languages, Information Structures and Machine Organization. McGraw-Hill, 1968.

[Weg 70] ______. Three Computer Cultures: Computer Technology, Computer Mathematics and Computer Science. Advances in Computers 10, 1970.

[Weg 71] ______. Data Structure Models for Programming Languages. Proc. ACM Symposium on Data Structures in Programming Languages, ACM SIGPLAN Notices, Feb. 1971.

[Weg 72a] ______. Programming Language Semantics, Formal Semantics of Programming Languages, ed. R. Rustin. Prentice-Hall, Englewood Cliffs, N.J., 1972.

[Weg 72b] ______. The Vienna Definition Language. ACM Computing Surveys 4, 1, March 1972, 5-63.

[Wir 66] Wirth, N., and Hoare, C.A.R. A Contribution to the Development of ALGOL. CACM 9, 9, Sept. 1966.

[Woz 69] Wozencraft, J.M., and Evans, A. Notes on Programming Linguistics. M.I.T. Department of Electrical Engineering, 1969.

Appendix

A MORE FORMAL TREATMENT OF BL

A.1. Interpreter States

An interpreter state embodies the information present at a given time in the computer system we are modeling. In this section we describe in detail the structure of BL-graphs representing interpreter states in the base language model. The treatment here differs somewhat from [Denn 71] and [Amer 72], but is essentially equivalent. In the next section we formalize BL-graphs and the BL instructions.

We assume that the reader is familiar with the concept of process as a locus of control. A process is represented in an interpreter state by a BL-object which we call a site of activity, or SOA. The BL-graph for an interpreter state is essentially a collection of SOA's. The root nodes of such a BL-graph are the root nodes of its SOA's. Thus an interpreter state is represented by a BL-graph whose skeletal form is shown in fig. A.1-1.

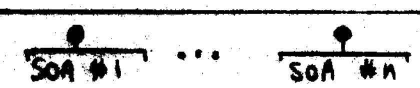


Fig. A.1-1. Skeletal structure of BL-graph for interpreter state

We now describe the structure of the individual SOA's of an interpreter state. A SOA is a BL-object with four components:

(1) The ep-component is a local structure, a BL-object representing the environment in which the SOA's computation takes place. (The name "ep" is an abbreviation for environment pointer.) Components of a local structure represent variables and temporaries used by the computation. Nearly all the BL instructions executed as part of the computation affect its local structure. We allow for the possibility of different SOA's sharing the same local structure, but usually the local structures of the different SOA's are distinct.

One distinguished SOA has as its ep-component a BL-object known as the universe. The universe represents the system-resident information present in the computer when no computations are in progress. Generally speaking, this information is independent of which computations are currently active or how far individual computations have progressed. This special SOA stands, so to speak, at the head of the system call chain, so that every process can trace its ancestry back to it. Access to the data in the universe is passed from caller to callee, so whatever access a particular SOA has to the universe is determined by the call chain leading back to the one distinguished SOA.

Two kinds of objects are found as components in the universe: data structures and procedure structures. Each kind of object can have objects of either kind as components. A data structure in the model can be any arbitrary BL-object; a procedure structure is a special kind of BL-object representing a procedure expressed in the base language. A BL instruction is easily represented as a BL-object; for example, the instruction const 3,x is depicted in figure A.1-2. The components with selectors 1,2,... of a procedure structure are simply representations of its instructions in order. A procedure structure may also have components which are procedure structures for nested procedures. Figure A.1-3 illustrates a skeleton procedure structure for a procedure p with one procedure f nested inside.

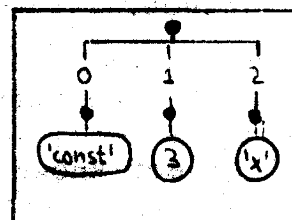


Fig. A.1-2. A sample BL instruction as a BL-object.

(2) The ip-component of a SOA gives the instruction currently being executed by the SOA's computation, as well as the procedure containing this instruction ("ip" stands

for instruction pointer). The ip-component is a two-component structure, whose proc-component gives the current procedure structure from which instructions are being executed, and whose instr-component gives the number of the instruction currently being executed in this procedure (fig. A.1-4). Thus the instruction currently being executed within a SOA *s* is given by the dotted pathname ip.proc.*(ip.inst), taken relative to the root node of *s*.

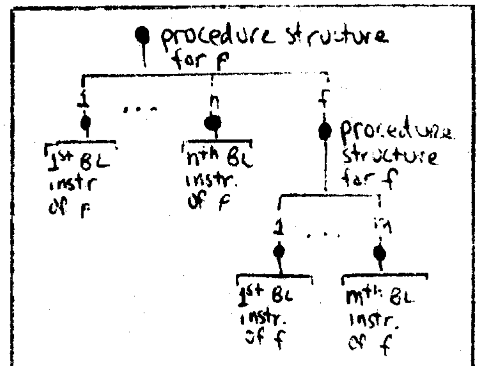


Fig. A.1-3. A sample procedure structure.

(3) The *stat*-component of a SOA, which gives its status, is an elementary object with the value 1 when the SOA is active (i.e. currently processing instructions), 0 if the SOA is dormant.

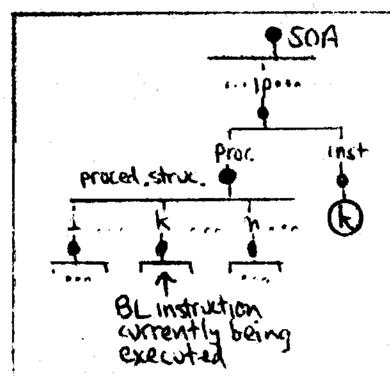


Fig. A.1-4. ip-component of a SOA

(4) The *ret*-component of a SOA *s* shares with the SOA that invoked (created) *s*. When *s* executes a *return* instruction, the SOA given by the ret-component of *s* is activated; the current SOA is put to sleep.

With the structure of an interpreter state given above, we can proceed to the next section, which describes how the BL instructions transform interpreter states.

A.2. BL-Graphs and BL Instructions

We give a formal mathematical definition of BL-graphs. Suppose the sets ELEM (elementary objects), SEL (selectors) and NODES (nodes) are given. For our purposes, ELEM shall consist of integers, truth values, real numbers and strings; SEL shall consist of integers and strings; and NODES shall be an arbitrary countably infinite set. Strings are taken over some suitable alphabet which includes the alphanumeric characters together with some special characters. A BL-graph over these three sets is a 4-tuple $g = (U,R,A,V)$ in which:

U (nodes in use)	is a finite subset of NODES;	
R (root nodes)	$\subseteq$	U;
A (arcs)	$\subseteq$	$U \times SEL \times U$;
V (valuations)	$\subseteq$	$U \times ELEM$.

We interpret $(\alpha,\sigma,\beta) \in A$ to mean there is a directed arc with selector σ leading from node α to node β; $(\alpha,\delta) \in V$ to mean α is a leaf node with elementary value δ. A BL-graph g must satisfy the following four conditions:

(1) If $\alpha \in U$, $\sigma \in SEL$, then there is at most one $\beta \in U$ for which $(\alpha,\sigma,\beta) \in A$.

(2) If $\alpha \in U$, then there is at most one $\delta \in ELEM$ for which $(\alpha,\delta) \in V$.

(3) $pr_1(A) \cap pr_1(V) = \emptyset$, where pr_1 is the first-component projection mapping. Equivalently,

$$\forall \alpha \in U: \sim[\exists\delta \in ELEM: ((\alpha,\delta) \in V) \; \& \; \exists(\sigma,\beta) \in SEL \times U: (\alpha,\sigma,\beta) \in A].$$

(4) $D^*(R) = U$, where D^* is the reflexive transitive closure of the immediate-descendant mapping $D: 2^U \rightarrow 2^U$ defined by

$$D(S) = \{\beta \in U: \exists\, \alpha \in S, \sigma \in SEL \text{ s.t. } (\alpha,\sigma,\beta) \in A\}.$$

Property (1) insures unique selection, i.e. that the selectors on the arcs emerging from a node are distinct. Property (2) asserts that no node may have more than one elementary value. Property (3) says that no node may have both components _and_ an elementary value, i.e. that elementary values can be attached only to leaf nodes. Property (4) states that every node of a BL-graph is accessible along some directed path of arcs starting with a root node.

We now give a formalism for defining transformations on BL-graphs. The formalism is based on [Denn 74]; it makes use of a set ID of identifiers and a mapping ν: ID $\cup$ ELEM $\cup$ NODES $\rightarrow$ ELEM $\cup$ NODES which assigns values to identifiers and acts as the identity function on elementary values and nodes. A <u>basic transformation</u> maps a BL-graph $g = (U,R,A,V)$ into a new graph $g' = (U',R',A',V')$ and updates the valuation mapping ν into a new mapping ν'. The notation $\nu[\alpha/x]$ means $\lambda y.(y{=}x \rightarrow \alpha,\ \underline{true} \rightarrow \nu(y))$, i.e. a mapping equivalent to ν except that it maps x into α.

The following basic transformations and auxiliary functions are defined for arbitrary BL-graphs:

<u>AddElem(a,d)</u>: [defined provided $\alpha \in U$, $\delta \in ELEM$, where $\alpha = \nu(a)$, $\delta = \nu(d)$]

$V' = V \cup \{(\alpha,\delta)\}$, $U' = U$, $R' = R$, $A' = A$, $\nu' = \nu$.

<u>DeleteElem(a,d)</u>: [defined provided $\alpha \in U$, $\delta \in ELEM$ and $(\alpha,\delta) \in V$, where $\alpha = \nu(a)$, $\delta = \nu(d)$]

$V' = V - \{(\alpha,\delta)\}$, $U' = U$, $R' = R$, $A' = A$, $\nu' = \nu$.

<u>AddArc(a,s,b)</u>: [defined provided $\alpha,\beta \in U$, $\sigma \in SEL$, where $\alpha = \nu(a)$, $\sigma = \nu(s)$, $\beta = \nu(b)$]

$A' = A \cup \{(\alpha,\sigma,\beta)\}$, $U' = U$, $R' = R$, $V' = V$, $\nu' = \nu$.

<u>DeleteArc(a,s,b)</u>: [defined provided $\alpha,\beta \in U$, $\sigma \in SEL$ and $(\alpha,\sigma,\beta) \in A$, where $\alpha = \nu(a)$, $\sigma = \nu(s)$, $\beta = \nu(b)$]

$A' = A - \{(\alpha,\sigma,\beta)\}$, $U' = U$, $R' = R$, $V' = V$, $\nu' = \nu$.

<u>DeleteComps(a)</u>: [defined provided $\alpha \in U$, where $\alpha = \nu(a)$]

$A' = A \cap ((U - \{\alpha\}) \times SEL \times U)$, $U' = U$, $R' = R$, $V' = V$, $\nu' = \nu$.

Prune:

$U' = D^*(R)$, $R' = R \cap U'$, $A' = A \cap (U' \times SEL \times U')$,

$V' = V \cap (U' \times ELEM)$, $\nu' = \nu$.

HasComp(a,s): [defined provided $\alpha \in U$, $\sigma \in SEL$, where $\alpha = \nu(a)$, $\sigma = \nu(s)$]

if $\exists \beta \in U$: $(\alpha,\sigma,\beta) \in A$ **then** true **else** false.

Comp(a,s) → b: [defined provided $\alpha \in U$, $\sigma \in SEL$ and HasComp(a,s) = true i.e. $\exists \beta \in U$: $(\alpha,\sigma,\beta) \in A$, where $\alpha = \nu(a)$, $\sigma = \nu(s)$]

let $\beta \in U$ **such that** $(\alpha,\sigma,\beta) \in A$;

$\nu' = \nu[\beta/b]$, $U' = U$, $R' = R$, $A' = A$, $V' = V$.

HasElem(a): [defined provided $\alpha \in U$, where $\alpha = \nu(a)$]

if $\exists \delta \in ELEM$: $(\alpha,\delta) \in V$ **then** true **else** false.

Elem(a) → d: [defined provided $\alpha \in U$ and HasElem(a) = true i.e. $\exists \delta \in ELEM$: $(\alpha,\delta) \in V$, where $\alpha = \nu(a)$]

let $\delta \in ELEM$ **such that** $(\alpha,\delta) \in V$;

$\nu' = \nu[\delta/d]$, $U' = U$, $R' = R$, $A' = A$, $V' = V$.

NewNode → a:

let $\alpha \in NODES - U$;

$\nu' = \nu[\alpha/a]$, $U' = U \cup \{\alpha\}$, $R' = R$, $A' = A$, $V' = V$.

MakeRoot(a): [defined provided $\alpha \in U - R$, where $\alpha = \nu(a)$]

$R' = R \cup \{\alpha\}$, $U' = U$, $A' = A$, $V' = V$, $\nu' = \nu$.

RemoveRoot(a): [defined provided $\alpha \in R \subseteq U$, where $\alpha = \nu(a)$]

$U' = U - \{\alpha\}$, $R' = R - \{\alpha\}$, $A' = A$, $V' = V$, $\nu' = \nu$.

The following transformations are composites of basic transformations:

```
NewComp(a,s) → b:
  NewNode → b;          [n.b. the semicolon indicates com-
  AddArc(a,s,b).         position of transformations, with
                         application in the order shown]

DeleteComp(a,s):
    if HasComp(a,s)          [the composite transforma-
      then {Comp(a,s) → b;    tion in the set braces is
            DeleteArc(a,s,b);  applied iff the node de-
            Prune}.            noted by a has a component
                               with selector denoted by s]

MakeEmpty(a,s) → b:               [makes b denote an empty
  if HasComp(a,s)                  leaf node which is the
    then {Comp(a,s) → b;           s-component of the node
          if HasElem(b)            denoted by a]
            then {Elem(b) → d;
                  DeleteElem(b,d)}
            else {DeleteComps(b);
                  Prune} }
    else NewComp(a,s) → b.
```

We now have the machinery to describe the action of the BL interpreter. The basic action is to pick a root node, which will be some SOA, then to execute the next instruction (given by the ip-component of the SOA) with respect to the current local structure (given by the ep-component of this SOA). Figure A.2-1 illustrates the skeletal structure of a sample SOA. In the procedure we will give to define the action of the interpreter, special names are used to designate nodes in the current SOA. These names appear as labels for the nodes in fig. A.2-1.

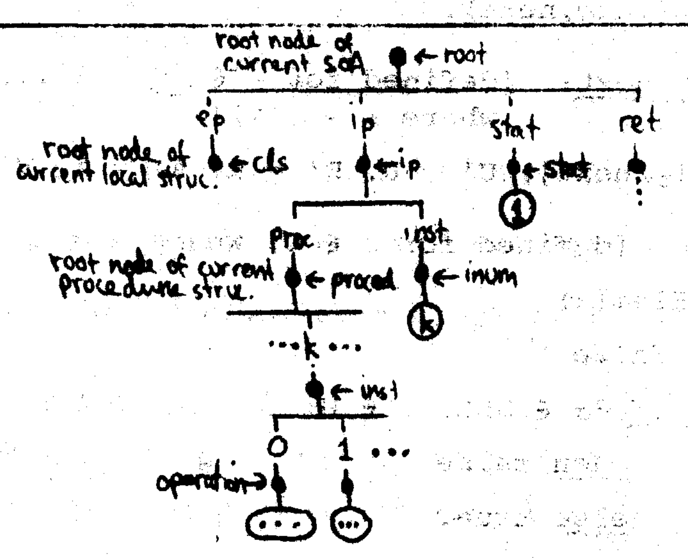


Fig. A.2-1. Structure of a SOA during interpretation.

Before giving a procedure which specifies the action of the BL interpreter, we define several auxiliary transformations. These use the special names shown in fig. A.2-1.

<u>PickActiveRoot → Root</u>:

<u>let</u> $\alpha \in R$ <u>such that</u> $\exists\beta \in U$: $(\alpha, \text{'stat'}, \beta) \in A$ & $(\beta, 1) \in V$;

$\nu' = \nu[\alpha/\text{root}]$, $U' = U$, $R' = R$, $A' = A$, $V' = V$.

<u>Succ → next</u>:

$\nu' = \nu[\kappa+1/\text{next}]$, $U' = U$, $R' = R$, $A' = A$, $V' = V$,

where $\kappa = \nu(k)$.

```
GetNextInstr:
  DeleteElem(inum,k);
  AddElem(inum,next).

Jump(i) → next:  [defined for ι ∈ {0,1,2,...} ⊆ ELEM,
                  where ι = ν(i)]
  ν' = ν[ι/next], U' = U, R' = R, A' = A, V' = V.

Empty(a):  [defined for α ∈ U, where α = ν(a)]
  if HasElem(a)
    then false
    else if ∃σ ∈ SEL, β ∈ U: (α,σ,β) ∈ A
           then false
           else true.
```

The action of the BL interpreter is specified by the repetitive application of the transformation given by the following procedure:

```
PickActiveRoot → root;         /* pick an active root node    */
Comp(root,'ep') → cls;         /* access the c.l.s. via ep    */
Comp(root,'ip') → ip;
Comp(ip,'proc') → proced;      /* access procedure structure */
Comp(ip,'inst') → inum;        /* number of current instr.    */
Elem(inum) → k;
Comp(proced,k) → inst;         /* fetch current instruction   */
Succ → next;                   /* set for next instruction    */
ExecuteBLInstruction(inst);    /* execute the instruction     */
GetNextInstr.                  /* reset ip for new instr.     */
```

Finally, we define the operation of all the BL instructions by giving the transformation ExecuteBLInstruction.

```
ExecuteBLInstruction(inst):
Comp(inst,0) → operation;
case operation of   /* choose the action that matches the
                       operation code of the instruction   */
  'create':
     Comp(inst,1) → x;                        /* create x     */
     DeleteComp(cls,x);
     NewComp(cls,x) → a.
  'clear':
     Comp(inst,1) → x;                        /* clear x      */
     MakeEmpty(cls,x) → a.
  'delete':
     Comp(inst,1) → x;
     if ¬HasComp(inst,2)
       then DeleteComp(cls,x)                 /* delete x     */
       else {Comp(inst,2) → m;                /* delete x,m   */
             if HasComp(cls,x)
               then {Comp(cls,x) → a;
                     DeleteComp(a,m)} }.
  'const':
     Comp(inst,1) → v;
     Comp(inst,2) → x;                        /* const v,x    */
     MakeEmpty(cls,x) → a;
     AddElem(a,v).
  'add':
     Comp(inst,1) → x;
     Comp(inst,2) → y;
```

```
    Comp(inst,3) → z;                          /* add x,y,z     */
    Comp(cls,x) → a; Comp(cls,y) → b;
    Elem(a) → d; Elem(b) → e;
    MakeEmpty(cls,z) → c;
    AddElem(c, ψ(d)+ψ(e)).
  .
  .     /* other arithmetic instructions are similar       */
  .
'link':
    Comp(inst,1) → x;
    Comp(inst,2) → n;
    Comp(inst,3) → y;                          /* link x,n,y    */
    Comp(cls,x) → a; Comp(cls,y) → b;
    if HasElem(a)
      then {Elem(a) → d; DeleteElem(a,d)}
      else DeleteComp(a,n);
    AddArc(a,n,b).
'select':
    Comp(inst,1) → x;
    Comp(inst,2) → n;
    Comp(inst,3) → y;                          /* select x,n,y */
    Comp(cls,x) → a;
    if ¬HasComp(a,n)
      then {if HasElem(a)
              then {Elem(a) → d;
                    DeleteElem(a,d)};
            NewComp(a,n) → b}
      else Comp(a,n) → b.
'apply':
    Comp(inst,1) → p;
```

```
   Comp(inst,2) → x;                          /* apply p,x      */
   Comp(cls,p) → proc; Comp(cls,x) → arg;
   Comp(proc,'$text') → t;
   NewNode → newsoa;
   NewComp(newsoa,'ep') → newcls;
   AddArc(newcls,'$par',arg);
   NewComp(newsoa,'ip') → newip;
   AddArc(newip,'proc',t);
   NewComp(newip,'inst') → newinum;
   AddElem(newinum,1);
   NewComp(newsoa,'stat') → newstat;
   AddElem(newstat,1);
   AddArc(newsoa,'ret',root);
   MakeRoot(newsoa);
   Comp(root,'stat') → stat;
   DeleteElem(stat,1); AddElem(stat,0).
'return':
   Comp(root,'ret') → oldsoa;
   Comp(oldsoa,'stat') → oldstat;
   DeleteElem(oldstat,0); AddElem(oldstat,1);
   RemoveRoot(root); Prune.
'move';
   Comp(inst,1) → f;
   Comp(inst,2) → x;                          /* move f,x       */
   Comp(proced,f) → a;
   DeleteComp(cls,x); AddArc(cls,x,a).
'goto':
   Comp(inst,1) → ℓ;                          /* goto ℓ         */
   Jump(ℓ) → next.
```

```
'elem?':
   Comp(inst,1) → x;
   Comp(inst,2) → ℓ;                           /* elem? x,ℓ       */
   Comp(cls,x) → a;
   if ¬HasElem(a)
     then Jump(ℓ) → next.
'empty?':
   Comp(inst,1) → x;
   Comp(inst,2) → ℓ;                           /* empty? x,ℓ      */
   Comp(cls,x) → a;
   if ¬Empty(a)
     then Jump(ℓ) → next.
'nonempty?':
   Comp(inst,1) → x;
   Comp(inst,2) → ℓ;                           /* nonempty? x,ℓ */
   Comp(cls,x) → a;
   if Empty(a)
     then Jump(ℓ) → next.
'eq?':
   Comp(inst,1) → x;
   Comp(inst,2) → y;
   Comp(inst,3) → ℓ;                           /* eq? x,y,ℓ       */
   Elem(x) → d; Elem(y) → e;
   if ν(d) ≠ ν(e)
     then Jump(ℓ) → next.
'has?':
   Comp(inst,1) → x;
   Comp(inst,2) → m;
```

```
    Comp(inst,3) → ℓ;                          /* has? x,m,ℓ     */
    if ¬HasComp(x,m)
      then Jump(ℓ) → next.
 'same?':
    Comp(inst,1) → x;
    Comp(inst,2) → y;
    Comp(inst,3) → ℓ;                          /* same? x,y,ℓ    */
    if ν(x) ≠ ν(y)
      then Jump(ℓ) → next.
 .
 .       /* other comparison instructions are similar        */
 .
 'getc':
    Comp(inst,1) → x;
    Comp(inst,2) → i;
    Comp(inst,3) → ℓ;                          /* getc x,i,ℓ     */
    Comp(cls,x) → a; MakeEmpty(cls,i) → b;
    if HasUnmarkedComps(a)
      then {GetUnmarkedComp(a) → s;
            Mark(a,s);
            AddElem(b,s)}
      else {UnmarkCompsOf(a);
            Jump(ℓ) → next}.
endcase
```

This completes the definition of the transformation ExecuteBLInstruction. The getc instruction, however, requires some special additional mechanisms, which we now show.

HasUnmarkedComps(a): [defined provided $\alpha \in U$, where $\alpha = \nu(a)$]

if $\exists\sigma \in SEL$: $(\alpha,\sigma,\beta) \in A$ **for some** $\beta \in U$

and $\sigma \notin MARKSET(\alpha)$

then true **else** false.

GetUnmarkedComp(a) → s: [defined provided $\alpha \in U$ and HasUnmarkedComps(a) = true, where $\alpha = \nu(a)$]

let $\sigma \in SEL$ be as in the HasUnmarkedComps predicate;

$\nu' = \nu[\sigma/s]$.

Mark(a,s): [defined provided $\alpha \in U$ and $\sigma \in SEL$, where $\alpha = \nu(a)$, $\sigma = \nu(s)$]

$MARKSET(\alpha) \leftarrow MARKSET(\alpha) \cup \{\sigma\}$.

UnmarkCompsOf(a): [defined provided $\alpha \in U$, where $\alpha = \nu(a)$]

$MARKSET(\alpha) \leftarrow \emptyset$.

We observe that each node $\alpha \in U$ has a set $MARKSET(\alpha)$ associated with it. All such marksets are initially empty.

There is one final remark to be made. Although our definitions of the BL instructions contain many composite transformations, the interpreter is to regard the effect of a BL instruction as an indivisible unit.