AN INTERPRETIVE MODEL FOR A LANGUAGE BASED ON SUSPENDED CONSTRUCTION*†

by

Steven D. Johnson

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 68

AN INTERPRETIVE MODEL FOR

A LANGUAGE BASED ON SUSPENDED CONSTRUCTION

STEVEN D. JOHNSON

*Submitted to the faculty of the Graduate School in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science, Indiana University.

†Research reported herein was supported (in part) by the National Science Foundation under grant number MCS75-08145.

An Interpretive Model For A Language Based On Suspended Construction

Steven Dexter Johnson

Abstract:

An interpreter for a purely applicative algorithmic language whose record constructor is based on suspended computation has been implemented in PASCAL on the CDC 6600. Suspended construction provides a model for the execution of algorithms in a multiprocessor environment without burdening the programmer with the scheduling of processes.

The syntax of the language is presented and a detailed discussion of the implementation establishes the semantics of the language. The presence of suspensions in the system and the applicative control structure make possible the creation of nondeterministically ordered data structure.

Submitted by:

____
Steven Dexter Johnson

____
Date

Accepted by:

____
Daniel P. Friedman

____
Date

____
David S. Wise

____
Date

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

Introduction

The programming language LISP [14] enjoys increasing popularity as a modeling tool and as an educational vehicle. Some of the reasons for this are obvious. Its generalized structure and symbol orientation, and automatic storage management relieve the programmer of the responsibility to maintain his data space. Its uniformity of program and data, interpretive execution, and informative error recovery facilitate debugging. Its applicative control structure induces modularity of thought and program.

With each of these benefits comes cost in both time and space and because of this, LISP and languages like it see little use beyond academic circles. Yet as the price of computation grows to depend ever more greatly on the human interface, and as hardware costs diminish, machine resources become less of an issue and demand increases for more expressive languages. Applicative languages are profoundly expressive, even though they lack the plethora of control structures found in current iterative languages.

This expressiveness comes from a close similarity to the language of formal mathematics, developed over the last several centuries without regard to the limitations of electronic technology. The research which led to this paper is motivated by the desire to create a natural, human oriented algorithmic language, and at the same time to address the issues of efficiency that confront the user of computers. "Efficiency" has two aspects: space and time. A reasonable decrease in space-efficiency can be absorbed by the shrinking cost of electronic components, but time is another matter; technology is nearing the theoretical limits for component speed while problems grow without end. Since applicative constructs do not reflect the physical architecture of computation devices, and iterative constructs do, it is not likely that applicative languages will ever be as fast, on the average, as iterative ones. But the gap can be narrowed. One approach to speeding computation is the collection of several computers for a single task. We see this happening now on a small scale with "smart" peripheral devices, vectored processors, and multiple CPUs. The semantics of function application lends itself readily to multiprocessing, and here lies its greatest promise at a practical level: there is an intuitive identity between process and processor.

In 1976 Friedman and Wise proposed a change in relationship between data structure <u>builders</u>, like LISP's CONStructor function, and structure <u>probes</u>, like CAR and CDR [5]. Under their scheme, the act of creating an instance of a structure involves no computation; the fields of the record are filled instead with transparent entities, called <u>suspensions</u>, which retain enough information to produce the correct sub-structure. Computation takes place when suspensions are accessed by the probes. As a result, compute effort is not expended on structures that aren't used. As a means for continued study, a model for this evaluation strategy was implemented in LISP in 1975. In the spring and summer of the next year, Cynthia Brown, then a graduate student, converted the model to PASCAL.

Late in 1976, I took responsibility for maintaining that program, an interpreter for a purely applicative language in which all computation is suspended. This paper, in part, is a report on the state of the program, which runs in interactive or batch mode on Indiana University's CDC6600 computer.

Chapter One is a cursory introduction to the interpreter's syntax. Familiarity with applicative programming is assumed. A more thorough introduction to the language is found in Appendix A, of which Chapter One is a review. Chapter Two contains a number of example programs demonstrating features of the language.

In Chapter Three the program itself is described in general terms. The details of the implementation are taken up later in Chapters Six and Seven. The third chapter also lays the groundwork for discussions of multiprocessing in Chapters Four and Five.

This paper has several purposes. It serves as a user manual for those interested in exploring the applicative approach to programming. Toward this end the sections of concern are Chapters One and Two and Appendix A. The interpreter's program is an ongoing project. For those inspired to take part in its further development, Chapters Three, Six and Seven provide detailed code documentation. These chapters contain parallel discussions of the program, with respect to general behavior, implementation specifics, and ideas for improvement. For example, the evaluator is summarized in Section 3.5, a rather involved example is traced through its execution in Section 6.5, and suggestions are made about changes in the evaluation strategy in Section 7.5. By making three passes at describing the program as a whole, the programmer should gain a better understanding of its overall behavior.

My interest in this project is fueled by the conviction that suspensions provide a way to implement large scale general purpose multiprocessors. From this point of view, the interpreter, as described in Chapter Three, is a model for such a machine. In Chapter Four, proposals are made concerning how suspensions are used to create a manageable system. In Chapter Five the architecture for a crude multiprocessor is described.

I would like to thank Jennifer Rae Deam for her indispensable assistance in preparing this paper.. I am grateful to Cynthia Brown for helping me to establish an understanding of the interpreter, to Don McKay, Bill White, and Tom Grismer; our frequent debates have fueled my enthusiasm for this topic. I am endebted to David Winkel and Frank Prosser for helping me overcome my fear of hardware, and to Stan Hagstrom, Paul Purdom for their insight into the issues presented in Chapters Four and Five. And finally, I would like to express my deep appreciation to Dan Friedman and David Wise for their guidance in my research, and to the National Science Foundation for funding my work (Grant #MCS75-08145).

Chapter 1. Survey of the Language

This chapter is an introduction to a programming language now being developed at Indiana University. The language is implemented with an interpreter, and is in some ways similar to the programming language LISP. Since it has no name yet, the language is referred to as the Interpreter. Appendix A is a more detailed discussion of the ideas presented here, written for those who are not familiar with applicative languages -- languages in which the only control structure is the application of functions to arguments. [12] Experienced LISP programmers will find that this chapter contains the information necessary for use of the Interpreter. Others should read Appendix A first and use this chapter for review.

The purpose here is not to teach how to write programs. There are other places where applicative programming skills can be learned; *The Little Lisper* [3] is an excellent place to begin. Interactive LISP interpreters are also helpful. The student must bear one inconvenience, though; LISP *syntax*, the form in which statements are made and programs are written, differs slightly from that of the Interpreter.

The language is data oriented, and data is divided into two categories. *Atoms* are elemental data consisting of *numbers* (integers only) and *literals* (finite length* character strings of digits and letters, starting with a letter). Data which are not atomic are called *ferns*; for the moment we are concerned with a restricted kind of ferns called *lists*. A list is an

---

*In the current implementation, numeric atoms are restricted to have an absolute value less than 65000, and literal atoms must have fewer than nine characters. Atoms of excessive length or magnitude are truncated by the Interpreter.

ordered collection of data. Lists are expressed by enclosing their elements in angle brackets, '<' and '>'. The empty list,<>, is called NIL.

X, ABC, and STEP5 are literal atoms.
6, -372, and 0 are numeric atoms.
<1 2 3>, <<2 3 5> 7> , and << >> are lists.

At the top level, the Interpreter is an implicit READ-EVALUATE-PRINT loop. Input to the program is a sequence of <u>forms</u> of which there are four kinds: atoms, lists, definitions, and applications. The syntax for definitional- and applicative forms is described below. The Interpreter evaluates each form in the input sequence, and returns a value:

form -=> value.

"-=>" is read "evaluates to". Numbers evaluate to themselves.

5 -=> 5
100 -=> 100
-1 -=> -1

If a list form is presented to the Interpreter, each element is evaluated and the result is the list of values. The Interpreter expresses lists by enclosing them in parentheses, '(', and ')'. The literal atoms FALSE and NIL evaluate to the empty list, and only one other literal, TRUE, has a value at the top level:

NIL -=> ().
FALSE -=> ().
TRUE -=> TRUE.
<> -=> ().
<1 2 3> -=> (1 2 3).
<TRUE <FALSE <>>> -=> (TRUE (() ())).

Lists enclosed in parentheses are called <u>pure data</u>. The Interpreter follows the convention that everything presented to input is evaluated. The user can suppress automatic evaluation by preceding forms with the QUOTE character, '"'. QUOTEd data should be pure:

```
            "AUTUMN -=> AUTUMN.
               TRUE -=> TRUE.
"(1 2 (X Y) ()) -=> (1 2 (X Y) ()).
```

<u>Applicative forms</u> have the syntax:

function : argument

The colon is a data transformation operator called APPLY; the form F:A might be read "APPLY the function F to the argument A." The function part of an applicative form may be a list (fern)*;when it is, the argument is treated as a rectangular array. Each element of the function-fern is APPLY'd to the corresponding column of the argument-fern [8]. In example (i) of Figure 1-1, the function part of the applicative form is the atom ADD1, the argument part is the number 1, and the form evaluates to the number 2. In example (ii) the function is PLUS and the argument is a list of two numbers. The result is the number 4. PLUS is called a "binary function"; it acts on two elements of its argument list (fern). In example (iii) the function PLUS is enclosed in brackets making

---

*The current implementation severely restricts second order application of functions. The form <F:A>:<G:B> is not allowed because the function part of the form contains an applicative form, F:A. Likewise, direct calls to EVAL are not available, unless the user has defined this function explicitly.

```
(i)    ADD1:1
        -=>2

(ii)   PLUS:<2 2>
        -=>4

(iii)  <PLUS>:<
       < 2  >
       < 2  > >
        -=> (4)

(iv)   <PLUS PLUS>:<
       < 2     2  >
       < #     3  >
       < 4     #  > >
        -=> (6 5)

(v)    <<PLUS PLUS> PLUS>:<
       << 1     2  >  3  >
       << 1     2  >  3  > >
        -=> ((2 4) 6)
```

Figure 1-1

the function part of the form a list. Because of this, the argument is treated as a two row by one column array; PLUS is applied to the only column. The result is the list (4). In example (iv), the function part has two elements, the argument array two columns, so the result is a two element list. The first element of the function part is APPLY'd to the leftmost column, the list (2 # 4). Hashmarks are <u>place-holders</u> in argument arrays; including them makes applicative forms more readable, but they are ignored by the Interpreter. Example (v) demonstrates that function-lists may have function-lists as elements. The argument array must reflect the nesting.

Binary functions, like PLUS in the examples above, may be given an argument structure of any length, but they act only on the first two positions. Thus, PLUS:⟨2 2⟩ -=> 4 and PLUS ⟨2 2 2 2⟩ -=> 4 and PLUS: ⟨PLUS:⟨1 1⟩ 2 TRUE⟩ -=> 4. In fact, it may be stated as a rule that:

ALL FUNCTIONS TAKE EXACTLY ONE ARGUMENT.

As we show above, the argument to a function can be a complex structure.

The system provides nine arithmetic functions: ADD1, SUB1, PLUS, DIFF, TIMES, DIV, MOD, GREAT, and LESS. ADD1 and SUB1 are "unary", they take a single Numeric argument; the rest are binary. PLUS, DIFF, TIMES, and DIV do integer addition, subtraction, multiplication,and division. MOD produces the remainder of division. LESS and GREAT are integer comparison predicates. The results of predicate functions are truth values, falsity is represented by the empty fern, ⟨ ⟩, or the Atom FALSE. All other structures may be said to represent truth.

```
PLUS:< TIMES:<10 10>  ADD1:1 >
 -=>102

DIFF:< 3  MOD:<4 3>>
 -=>0

GREAT:< DIV:<3 5>  -1>
 -=>TRUE

LESS:< MOD:<3 4> -333>
 -=>()
```

Figure 1-2

Other system functions are used for the examination of data structures and are very familiar to LISP users. The unary predicate ATOM returns TRUE when its argument is atomic. The unary predicate NULL (or NOT) is a test for the empty fern, $\langle\rangle$. The binary predicate SAME is used to compare Atoms* and returns true if the first two elements of its argument are identical. The functions FIRST (similar to LISP's CAR) and REST (CDR) take a list (fern) argument. FIRST returns the first element of the list and REST returns what remains of the list when the first element has been removed. Finally, a number may appear in the function position of an applicative form. <u>Numeric</u> <u>functions</u> are shorthand notation for often-used combinations of FIRST and REST. If the number is <u>i</u>, it stands for <u>i</u>-1 calls to REST and one call to FIRST.

```
FIRST:<1 2 3>
 -=>1

REST:<1 2 3>
 -=> (2 3)

FIRST:FIRST:REST:<1  <2 3> <4>>
 -=>2

FIRST:REST:REST:<1 2 3 4 5>
 -=>3

1:<1 2 3>
 -=>1

2:<1 2 3>
 -=>2

3:<1 2 3>
 -=>3
```

Figure 1-3

In the current implementation, SAME may be used to compare list structures. It returns TRUE when both argument-elements are , and FALSE <u>in all other cases</u>. Thus, SAME is not LISP's EQ.

The last kind of form that the Interpreter accepts is a definitional form, used for the definition of functions and the declaration of program constants. Definitional forms are the only means of adding information to the Interpreter's top-level data base. The syntax is:

DEFINE function_name formal_parameter function_body.

DECLARE constant_name value.

The period is part of these forms. Function_names and constant_names are literal atoms.

When the Interpreter is given a constant declaration it evaluates the value part of the form, then binds the constant_name to the result in the top-level environment. Constants may not be re-DECLAREd.

The formal_parameter part of a function definition is a pattern for the argument the function accepts, expressed as pure data. The function_body is a statement of what the value of the function is when it is APPLY'd to a particular argument. The Interpreter finds this value by "executing" the function_body, that is, the function_body is evaluated, and the result is returned as a value for the application.

When it is given a function definition, the Interpreter binds the function_name to the parameter-body pair in the top-level environment. At application time, the formal_parameter is retrieved and compared with the actual argument in the applicative form. During comparison, substructures of the formal_parameter are matched in the argument, and the formal_parameter's atoms are bound to the data in the corresponding positions of the argument. Then the function_body is evaluated in this newly created environment. The binding operation is actually suspended, or postponed by the Interpreter;

later, if values are needed during evaluation of the function_body, binding resumes. This is described in more detail below, but one result of suspending parameter binding is that the lengths of the argument and the formal_parameter are never compared -- recall that binary arithmetic functions can be given too many argument-elements -- therefore, it is always possible to give too large or too complex an argument to a user defined function.

```
DEFINE ADD2  X  ADD1:ADD1:X.
 -=>ADD2

ADD2:4
 -=>6


DEFINE MEAN (X Y)  DIV:<PLUS:<X Y> 2>.
 -=>MEAN

MEAN:<23 10>
 -=>16


DEFINE RESTRUCT ((X Y) Z ((W)))  <W X Y Z>.
 -=>RESTRUCT

RESTRUCT:<<2 3> 4 <<1>>>
 -=> (1 2 3 4)


MEAN:RESTRUCT:<<2 3> 4 <<8>>>
 -=>5
```

Figure 1-4

The function_body is a conditional expression, a list containing an odd number of forms. Odd-numbered forms, except for the last, are taken to be conditional predicates which are evaluated in the order that they appear in the definition. The first of these predicates to evaluate to other than FALSE (or NIL, or ()) causes the next (even-numbered) form to be evaluated and returned as a result.* In case all of the predicates

*There is no LISP-like CONDitional function, rather, conditional behavior is implicit in function execution. The nesting of conditional expressions is done with auxiliary functions.

fail, the last form is evaluated and returned. If no final alternative is given in the definition, that is, if the user defines a function with an even-length list of forms, and if all the predicates fail, then () is returned by the Interpreter.

```
DEFINE SIGMA LISTNUMS
       IF NULL:LISTNUMS  THEN 0
     ELSE PLUS:<FIRST:LISTNUMS  SIGMA:REST:LISTNUMS>.
 -=>SIGMA

SIGMA:<1 2 3 4 5>
 -=>15


DEFINE BSEARCH  (KEY (ROOT INFO LSUBTREE RSUBTREE))
       IF SAME:<ROOT 0>  THEN "(ENTRY NOT IN TREE)
   ELSEIF SAME:<ROOT KEY> THEN INFO
   ELSEIF LESS:<ROOT KEY> THEN BSEARCH:<KEY LSUBTREE>
     ELSE BSEARCH:<KEY RSUBTREE>.
 -=>BSEARCH

BSEARCH:<  2
         <  3  "A
            <4   "B  <0>  <0>>
            <2   "C  <0>  <0>>    >>
 -=>C
```

Figure 1-5

In example (ii) of Figure 1-5, a binary search algorithm, the structure of the search tree is specified in the formal parameter; it is a list containing a ROOT, the INFOrmation associated with the root node, and a Left- and Right-SUBTREE. BSEARCH is applied to a tree whose ROOT is 3; the root node contains the information "A. The subtrees of this structure are also binary trees. For example, the left-

SUBTREE has ROOT 4 and information "B; its subtrees are empty. The empty subtrees are incomplete structures, containing a ROOT only. If BSEARCH is applied to such a tree, the first alternative succeeds, ROOT is zero, so the value returned is (ENTRY NOT IN TREE). INFO, LSUBTREE, and RSUBTREE are not included because when the ROOT is zero, these variables are not used by function body. Since they are not used, they are never bound; since they are not bound they need not exist.

We have not yet discussed the most important functions of all, the constructors. Constructors are the primary tools for structure oriented problems. The LISP constructor, CONS, takes two arguments, evaluates them, and creates a list out of the results. The FIRST element of this new list is the value of the first argument; the REST of the new list is the value of the second argument. The Interpreter has two constructors, CONS and FONS. As its name indicates, CONS is a relative of the LISP constructor; CONS is used to create lists. FONS creates an unordered structure, called a multiset. Lists, multisets, and mixtures of the two make up the class of structures called ferns.

We discuss CONS first. Unlike the LISP's CONS, the Interpreter's CONS does not evaluate its arguments. Instead, the FIRST and REST fields of the new cell contain transcient structures called suspensions. A suspension is a promise to evaluate a form whenever it becomes necessary to do so [5]. Applying the list probes, FIRST and REST, to suspensions forces the evaluation to take place. Suspensions are never seen by the user; if she tried to look at them, by calling FIRST or REST, evaluation would be forced, the result would be returned instead.

Example:

DEFINE INTEGERS N CONS:⟨N INTEGERS:ADD1:N⟩ .

If the Interpreter encounters the applicative form, INTEGERS:1, the execution of the function body, CONS: ⟨. . .⟩ , causes a new list to be created. This list contains two suspensions (Figure 1-6).

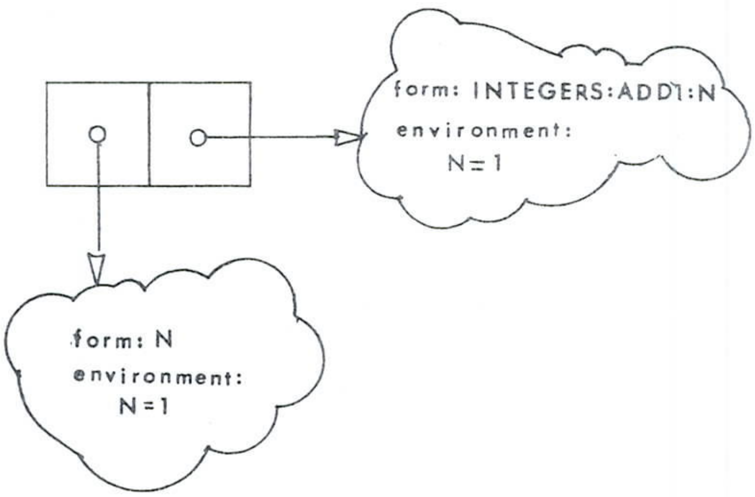


Figure 1-6

The suspensions are shown as clouds in the figure. Suppose that sometime later, REST were applied to this list. The righthand suspension is <u>coerced</u> (evaluated); it contains the form INTEGERS:ADD1:N, which means another call to CONS is necessary. The suspension is replaced by a second new list cell which contains two more suspensions (Figure 1-7).

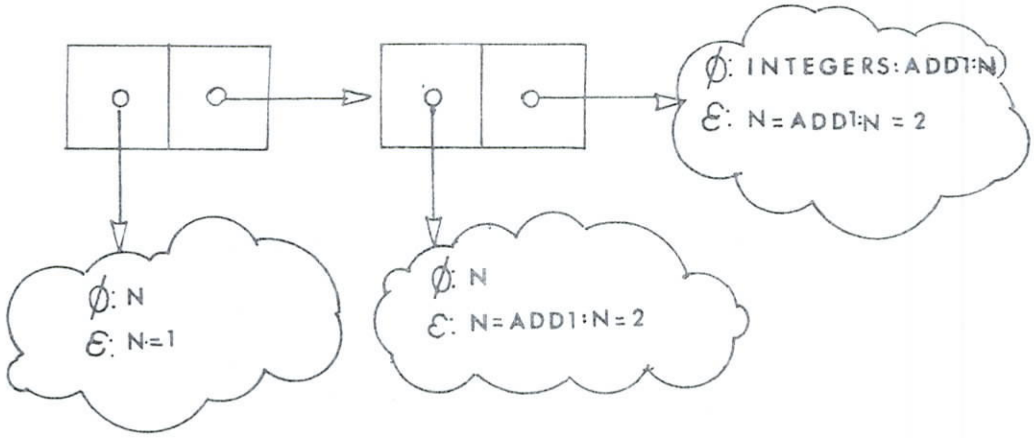


Figure 1-7

Now if FIRST is called on the second cell, this probe discovers a suspension whose form is N. In the suspension's environment, the atom N is bound to the form ADD1:N (another suspension). The second occurrence of the atom N is bound in the first cell's environment to the number 1. ADD1:N evaluates to 2, and this value replaces the suspension in the FIRST field of the second cell (Figure 1-8).

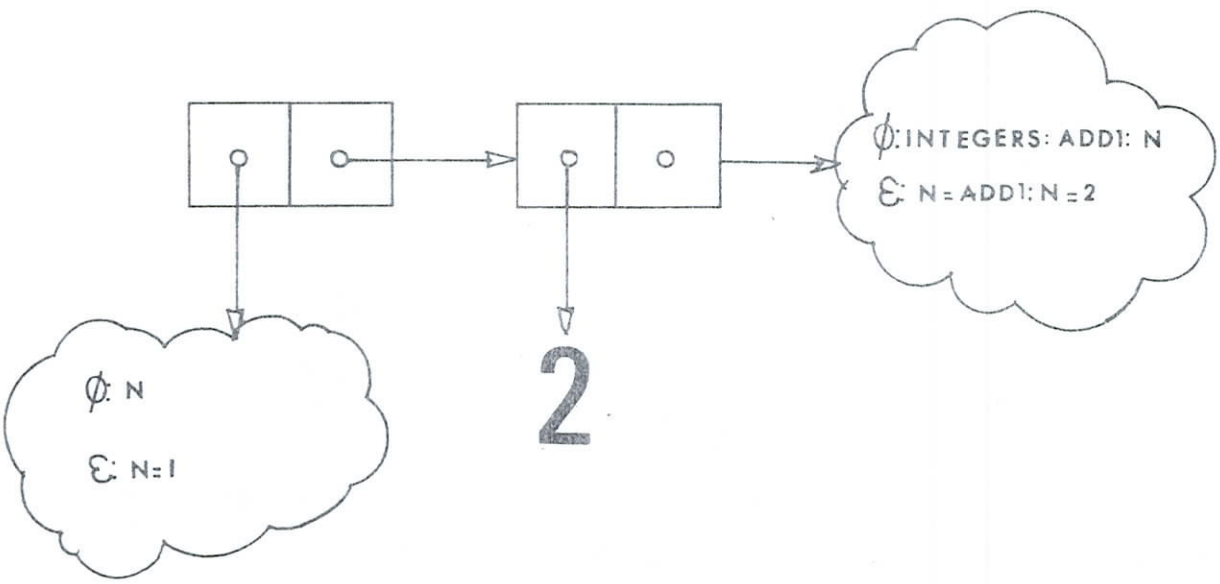


Figure 1-8

The Interpreter replaces suspensions with their values so that repeated probes on a cell won't cause repeated evaluation.

Two generalizations on a pure-LISP Interpreter are immediate: since the only necessary calls to suspension-coercing probes result from the need to print answers on the output device, only computation essential to finding those answers is carried out. The computer does a minimum of work. Second, suspensions make it possible for the Interpreter to manipulate structures which appear to be infinitely large. The list created by INTEGERS:1 grows longer with every call to REST. A standard LISP Interpreter fails to build even the first cell of INTEGERS:1. LISP's constructor evaluates both arguments before fetching a cell, (unsucessfully in this case).

Multisets are "unordered" when they are created. FONS too, does not evaluate its arguments, but turns them into suspensions to be coerced by the probes. When the probes evaluate suspended elements of a multiset they select an order for the structure at the same time. In short, lists are ordered at construction time, multisets are ordered at access time. Consider the form:

FONS:⟨ADD1:1 FONS:⟨ADD1:ADD1:1 FONS:⟨1 ⟨⟩⟩⟩⟩

Evaluation of this form creates a multiset of three suspended elements. If FIRST probes this structure it coerces all of the suspensions at the same time. The first suspension to converge becomes the FIRST element of the structure. The form can evaluate to any of the lists:

(1 2 3), (1 3 2), (2 1 3),
(2 3 1), (3 1 2), or (3 2 1).

Now consider the form FONS:⟨ADD1:TRUE FONS:⟨7 ⟨⟩⟩⟩ . Evaluation of this form results in a multiset of two suspended elements, but one of the suspensions contains an illegal form. A second property of multisets is that erroneous or divergent computations are forced away from the beginning of the fern toward the end. Because ADD1:TRUE is undefined, when the multiset is probed the element 7 appears first. Thus, a probe on a multiset returns a value as long as the multiset contains at least one convergent element.

The user has a shorthand notation for calls to CONS: the angle bracket notation.

⟨1 2 3⟩ = CONS:⟨1 CONS:⟨2 CONS:⟨3 ⟨⟩⟩⟩⟩-=⟩(1 2 3).

Square brackets* are shorthand for FONS:

[1 2 3] = FONS:⟨1 FONS:⟨2 FONS:⟨3 ⟨⟩⟩⟩⟩-= (1 2 3) or (1 3 2) . . .

Finally, some special characters are reserved for use by the system. A few have been mentioned already. The double quote supresses automatic evaluation. The hashmark is a place holder in argument arrays. The colon is the APPLY operator in applicative forms. A star is used to denote infinite structures:

---

*In the works of Friedman and Wise, and in future versions of the Interpreter, square brackets are replaced by braces, "{" and "}". Square brackets were chosen partly because of the mapping of the ASCII braces (lower case characters) to the 64 character ASCII subset.

⟨7*⟩ -=⟩ (7*), which means (7 7 7 7 . . .).
⟨5 6 7*⟩ -=⟩ (5 6 7*), which means (5 6 7 7 7 . . .).
⟨PLUS*⟩:⟨
⟨1 2 3⟩
⟨1 2 3⟩ ⟩ -=⟩ (2 4 6)
⟨PLUS*⟩:⟨
⟨2*⟩
⟨2*⟩ ⟩ -=⟩ (4*)

Stars may be used in the function part of an applicative form; the Interpreter applies the starred function as many times as there are columns in the argument array.

The slash character has two uses. When multisets are built using the square bracket, a slash indicates that the preceeding element is to be CONSed into the structure. CONSed elements act as a fence in multisets; probes cause no evaluation beyond these fences until their suspensions have converged.

[1 2/ 3] = FONS:⟨1 CONS:⟨2 ⟨3⟩⟩⟩ -= (1 2 3) or (2 1 3) or (2 3 1).

[A/ B/ C/ D/] = ⟨A B C D⟩.

No form may contain two consecutive slashes, '//'. If this string appears in the input stream, the Interpreter rejects the form it is building, skips to a new line, and starts again. If a typographical error is noticed during a lengthy definition, typing two slashes is a way to start over. The Interpreter prompts the interactive user for more input with a question mark, '?'. A semicolon, ';', in the input stream causes the Interpreter to skip to the next line for more characters; this provides a way to add comments to programs. When the user is at top level, the special form "EXIT.", causes the Interpreter to halt, returning control to the host system.

REVIEW OF CHAPTER 1

1. Data is of type atom or fern. Atoms may be literals or numerics. Ferns may be lists or multisets.

2. The Interpreter is a READ-EVALuate-PRINT loop.

3. Acceptable forms:
   a. Literals and numerics. Numerics evaluate to themselves. TRUE -=> TRUE; NIL -=> (); FALSE -=> ().
   b. Ferns. Angle brackets enclose list structures; square brackets enclose multisets. A fern evaluates to a fern of evaluated elements.
   c. Applicative forms. The syntax is function:argument. The function part contains no applicative forms, the argument part may have any complexity. Fern-functions are applied to the columns of the argument array.
   d. Definitional forms. Function definitions have the syntax:
      DEFINE name formal-parameter body.

      Constant declarations have the syntax:
      DECLARE constant value.

      The body of a function is a list of forms separated by the keywords IF, THEN, ELSE, and ELSEIF.

4. The system provides functions for arithmetic operations and data manipulation.
   a. Unary arithmetic functions: ADD1, SUB1.
   b. Binary arithmetic functions: PLUS, DIFF, TIMES, DIV, MOD.
   c. Binary comparison predicates: LESS, GREAT, SAME.
   d. Data examination predicates: ATOM, NOT, NULL.
   e. Data probes: FIRST, REST, numeric functions.
   f. Data constructors: CONS, FONS.

5. All evaluation is suspended by the constructors.

6. Special purpose characters:
   a. "." -- end of form; end of definition.
   b. "#" -- place holder in argument arrays.
   c. "?" -- The Interpreter's prompt character.
   d. ";" -- comment, rest of line is ignored.
   e. "/" -- insert a list cell into this multiset.
   f. "//" - reject the current form in input.
   g. ":" -- the APPLY operator.
   h. '"' -- bypass automatic evaluation, "XYZ -=> XYZ.

7. The atom, EXIT, when evaluated, causes The Interpreter to stop, and returns the user to the system monitor.

Chapter 2. Some Demonstration Programs

Section 2.1 Examples

My primary responsibility as a research assistant for Dan Friedman and David Wise was to maintain an interpretive program for the language presented in Chapter One. The interpreter is a means to demonstrate their ideas about computation, about programming languages and style. As the language evolved and the semantics of constructors was embellished, the program has been altered; its behavior is an objective verification of their ideas as well as a tool for continued development of the applicative approach to computation.

In this chapter a number of examples are given to elucidate features of the language. All were run on Indiana University's Control Data Corporation 6600 computer under the KRONOS operating system. Most of the examples were submitted from a terminal in batch mode, with the source program on file, but they could have been executed interactively. The interpreter requires a core field length of 30,000 octal words; this includes an array of 7,000 cells. The programmer can, upon entry to the program, restrict the number of cells that the interpreter can use by supplying a size parameter in answer to the prompt:

-=> MEMORY LIMIT?

Resources are limited in some of the examples to show how much space some algorithms use.

Example 2.2 Suspended Construction

The original motivation for writing the interpreter was to create a system in which the list builder, CONS, does not evaluate its arguments but creates its result with no computational effort. The fields of the new record are filled instead with suspensions which serve as proxies for the eventual results [5]. One consequence of this approach to construction is that potentially infinite structures, a list of all the positive integers for example, can be defined and built (made manifest). The list is brought into existence not by the act of creation, but by the need for its contents in further computation. In this section, such lists are built; "further" computation results from using structure to write its contents on the output file.

The interpreter's storage reclamation system is based on a discrete reference count scheme; every data cell has a field set aside in which references to that cell are tabulated. Cells are returned to available space when their last reference is removed. Because the last reference to the lists in these examples is made by the PRINT routine, the structures are comsumed and returned as they are traversed [7], and the lists are created, printed, and returned in constant space. To demonstrate this, a list of integers is printed in Figure 2.2-1 while memory is constrained to 350 cells. Program initialization uses about 300 cells leaving 50 for the construction of the infinite list. The program is allowed to print enough of this list to show that no cells are lost by creation. In addition, concurrent storage reclamation enables the list to be traversed and disposed of without any computational pause for garbage collection [11, 2.3.5].

The example in Figure 2.2-2 shows that the memory constraint actually works by making it impossible for the interpreter to compute the next integer. The INTEGERS algorithm runs in about 330 cells which means that around twenty cells are used to create,suspend, and evaluate each element of the list structure.

The consumption of an infinite structure by printing depends on two things: there must be no additional references to the printed structure from within the interpreter, and there must be no recursive "buildup" of environments maintained in un-coerced suspensions. Figure 2.2-3 shows a program in which a computationally simpler structure than INTEGERS is constructed. The print algorithm is unable to recycle this structure completely (the list structure itself _is_ returned to available space), because the unused formal parameter's binding is never coerced into existence. The atom _n_ is bound in each recurrence to the number 1, but the binding is suspended and the suspension contains a reference to the value of _n_ in the previous environment. The excess baggage is carried along with each recursive call to ONESTAR , even though it is never to be used, and because of this memory is totally consumed.

The purpose of this example is to show that suspensions consume space. The problem of un-coerced arguments in this algorithm problem is solved when the interpreter is made sensitive to tail recursive forms [16].

```
 ---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT
? 350

? define INTEGERS   n   cons:< n  integers:add1:n >.

 -=>INTEGERS
? integers:1.

 -=> (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
  50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 7
 3 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96
 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 11
 5 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 1
 33 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150
 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168
  169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 18
 6 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 2
 04 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221
 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
  240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 25
 7 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 2
 75 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292
 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310
  311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 32
 8 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 3
 46 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363
 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381
  382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 39
 9 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 4
 17 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434
 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452
  453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 47
 0 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 4
 88 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505
 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523
  524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 54
 1 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 5
 59 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576
 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594
  595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 61
 2 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 6
 30 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647
 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665
  666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 68
 3 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 7
 01 702 703 704 705 706 707 708 709 710 711 712
 *TERMINATED*
```

Figure 2.2-1

```
 ---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT
? 320

? define INTEGERS  n  cons:< n  integers:add1:n >.

 -=>INTEGERS
? integers:1.

 -=> (1

 -=>-=>-=> MEMORY IS EXHAUSTED.
     -=> YOU HAVE SPECIFIED 320 NODES,
AND THE LIMIT IS 7000.
```

Figure 2.2-2

```
 ---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT
? 2000

? define ONESTAR  (n)  cons:< 1  onestar:<1> >.

 -=>ONESTAR
? onestar:<1>.

 -=> (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1

 -=>-=>-=> MEMORY IS EXHAUSTED.
     -=> YOU HAVE SPECIFIED 2000 NODES,
AND THE LIMIT IS 7000.
```

Figure 2.2-3

Example 2.3 General Application

Friedman and Wise propose that a construct be added to the vocabulary of applicative programming which enables the application of a functional structure to a suitably vectorized matrix [8]. This construct, called functional combination and later general application, provides a framework for concurrent computation in a multiprocessing system (see Chapter 4), as well as a stylized way to express non-linear recursive algorithms.

One benefit of stylization is that a reduction, or compilation of standard forms is made easier, and hence the reduction of applicative algorithms to optimal machine dependent ones.

Figure 2.3-1 shows an implementation of the Quicksort Algorithm as a recursive example of general application. Only the top level routine SORT is non-linear. Since all construction, including generally applicative forms, is suspended the sorting is driven by the need to print the result. If in the program, only the first element of the sorted list had been requested, a minimal amount of computation would have taken place, enough to find the smallest element, leaving the rest of the result structure un-coerced.

```
 ---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT


 -=> (TRACE 1)

DEFINE QSORT  (LIST)
      IF NULL:LIST THEN <>
 ELSE APPEND3:<QSORT 1 QSORT>:<SHUFFLE:< FIRST:LIST LIST>>.
 -=>QSORT


DEFINE SHUFFLE (HEAD TAIL)
      IF NULL:TAIL THEN <<> <> <>>
  ELSEIF GREAT:<HEAD FIRST:TAIL> THEN < CONS       1        1  >:<
                                      <1:TAIL     #        #  >
                                      SHUFFLE:<HEAD REST:TAIL> >
  ELSEIF LESS:<HEAD FIRST:TAIL>  THEN <  1        1       CONS >:<
                                      <  #        #      1:TAIL>
                                      SHUFFLE:<HEAD REST:TAIL> >
    ELSE <  1       CONS      1   >:<
         <  #      1:TAIL     #   >
        SHUFFLE:<HEAD REST:TAIL> >.
 -=>SHUFFLE

DEFINE APPEND3 (A B C)  APPEND:<A APPEND:<B C>>.
 -=>APPEND3

 DEFINE APPEND (A B)
       IF NULL:A THEN B
     ELSE CONS:<FIRST:A APPEND:<REST:A B>>.
 -=>APPEND


SHUFFLE:<4 <1 8 2 4 1 1 4 2 8>>.
 -=> ((1 2 1 1 2) (4 4) (8 8))

 QSORT:<<3 4 1 6 7 2 8 5 12 11 10 9>>.
 -=> (1 2 3 4 5 6 7 8 9 10 11 12)

 EXIT.
-=>-=>-=>-=> LEAVING.
NODES DISPOZED, 94931
NODES RECYCLED, 8083
AVAIL--> 1225
/
```

Example 2.4 Infinite Structures

The syntax of the language includes explicit fern structures of infinite length, implemented as circular lists. This example, creation and partial enumeration of the rational numbers, exercises that construct. Starred structures [8], may appear in the argument position of applicative forms, or as generally applicative functionals. Figure 2.4-1 shows both cases (see the function PROD1).

```
 ---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT


 -=> (TRACE 1)

DEFINE INTEGERS N CONS:<N INTEGERS:ADD1:N>.
 -=>INTEGERS

DEFINE PRODUCT (L1 L2) <PROD1*>:<L1 <L2*>>.
 -=>PRODUCT

DEFINE PROD1 (A L)  <ORDPAIR*>:<<A*> L>.
 -=>PROD1

DEFINE ORDPAIR (X Y) <X Y>.
 -=>ORDPAIR

DEFINE RATNLS NIL PRODUCT:<INTEGERS:1 INTEGERS:1>.
 -=>RATNLS

DEFINE APPEND (L1 L2)
      IF L1 THEN CONS:< FIRST:L1 APPEND:<REST:L1 L2>>
    ELSE L2.
 -=>APPEND

DEFINE SLICE (N L)
      IF SAME:<N 0> THEN <>
    ELSE CONS:<1:1:L SLICE:<SUB1:N REST:L>>.
 -=>SLICE

DEFINE SHAVE (N L)
      IF SAME:<N 0> THEN L
    ELSE CONS:<REST:FIRST:L SHAVE:<SUB1:N REST:L>>.
 -=>SHAVE

DEFINE ENUMRATE (N L)
      APPEND:<SLICE:<N L> ENUMRATE:<ADD1:N SHAVE:<N L>>>.
 -=>ENUMRATE

ENUMRATE:<1 RATNLS:<>>.
 -=> ((1 1) (1 2) (2 1) (1 3) (2 2) (3 1) (1 4) (2 3) (3 2) (4 1) (1 5)
 (2 4) (3 3) (4 2) (5 1) (1 6) (2 5) (3 4) (4 3) (5 2) (6 1) (1 7) (2 6)
  (3 5) (4 4) (5 3) (6 2) (7 1) (1 8) (2 7) (3 6) (4 5) (5 4) (6 3) (7 2
 ) (8 1) (1 9) (2 8) (3 7) (4 6) (5 5) (6 4) (7 3) (8 2) (9 1) (1 10) (2
  9) (3 8) (4 7) (5 6) (6 5) (7 4) (8 3) (9 2) (10 1) (1 11) (2 10) (3 9
 ) (4 8) (5 7) (6 6) (7 5) (8 4) (9 3) (10 2) (11 1) (1 12) (2 11) (3 10
 ) (4 9) (5 8) (6 7) (7 6) (8 5) (9 4) (10 3) (11 2) (12 1) (1 13) (2 12
 ) (3 11) (4 10) (5 9) (6 8) (7 7) (8 6) (9 5) (10 4) (11 3) (12 2) (13
 1) (1 14) (2 13) (3 12) (4 11) (5 10) (6 9) (7 8) (8 7) (9 6) (10 5) (1
 1 4) (12 3)
 *TERMINATED*
```

Figure 2.4-1

Example 2.5 Multisets

The interpreter provides two constructors for the creation of ferns. CONS is analogous to the LISP CONS, except that it suspends its arguments; the resulting list structure is ordered when it is built. The second constructor, FONS, builds more general structures called multisets, whose order is determined later, as its elements are evaluated [10]. Multisets carry the idea of suspensions another step. In essence the elements of a multiset compete for priority (Sections 3.7, 6.8), when the fern is probed. Thus, multisets can be used for modeling real time program behavior.

This example, Figure 2.5-1, shows the elements of the ordering strategy. The function LASTANGO takes two ferns and associates their elements pair-wise. To provide a measure of computational cost to the evaluation of the elements, they are embedded in a small relational network. LASTANGO is given two multisets to match up and begins recursively to fetch elements from them. The probes cause simultaneous evaluation of each element of each multiset; those which converge first are returned first. The result is a quasi-nondeterministic fern of element pairs.

```
 ---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT


 -=> (TRACE 1)

DEFINE LASTANGO (L1 L2)
      IF SAME:<L1 L2> THEN <<"IS "EVERY "BODY "HAPPY>>
  ELSEIF NULL:L1 THEN <<"WALL "FLOWERS L2>>
  ELSEIF NULL:L2 THEN <<"WALL "FLOWERS L1>>
    ELSE FONS:<[FIRST:L1 FIRST:L2] LASTANGO:<REST:L1 REST:L2>>.
 -=>LASTANGO

DEFINE BROTHER PERSON
       IF SAME:<PERSON "SAM> THEN "TONY
   ELSEIF SAME:<PERSON "TONY> THEN "SAM
   ELSEIF SAME:<PERSON "JILL> THEN "DAVE
     ELSE "DAN.
 -=>BROTHER

DEFINE SISTER PERSON
      IF SAME:<PERSON "MARY> THEN "CLEO
  ELSEIF SAME:<PERSON "CLEO> THEN "MARY
  ELSEIF SAME:<PERSON "DAVE> THEN "JILL
    ELSE "RITA.
 -=>SISTER

BROTHER:BROTHER:BROTHER:"SAM.
 -=>TONY

BROTHER:SISTER:"DAVE.
 -=>DAVE

SISTER:BROTHER:"TONY.
 -=>RITA

SISTER:"CLEO.
 -=>MARY

LASTANGO:
<[BROTHER:SISTER:"DAVE  BROTHER:BROTHER:BROTHER:"SAM   "DAN ]
 [SISTER:BROTHER:"TONY   "CLEO            SISTER:"CLEO       ]>.
 -=> ((CLEO DAN) (MARY TONY) (DAVE RITA) (IS EVERY BODY HAPPY))

LASTANGO:
   <["DAVE "TONY "DAN]
    ["RITA "CLEO "MARY]>.
 -=> ((DAVE RITA) (TONY CLEO) (DAN MARY) (IS EVERY BODY HAPPY))

EXIT.
-=>-=>-=>-=> LEAVING.
NODES DISPOZED, 4017
NODES RECYCLED, 614
AVAIL--> 632
/
```

Figure 2.5-1

Example 2.6 More Multisets

An immediate result of the FONS constructor is that it allows breadth-oriented tree searches as a primitive operation. The program in Figure 2.6-1 demonstrates this by solving a simple maze problem, and at the same time shows why simplistic approaches to problem solving should be avoided. The maze is expressed as a 3 x 4 matrix of truth values, declared to be a global constant. A starting position and a goal position are given to the top level function MAZE, which creates a multiset of four new states by changing the starting position. MAZE is applied to each new state, and the process continues until the goal is reached. Moves beyond the boundaries of the maze become undefined. If a position on the board is TRUE a move cannot be made there.

Murphy's Law, and my experience with the program lead me to estimate that roughly 150 branches (including the undefined ones) could be generated in the process of finding a path by blind, breadth-first searching in the third call to MAZE in Figure 2.6-1; the interpreter hasn't enough memory for that many, even by conservative estimates of the cost of suspensions, so it runs out of cells.

```
 ---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT


 -=> (TRACE 1)

DECLARE BOARD <<FALSE  FALSE  FALSE  FALSE >
               <FALSE  TRUE   FALSE  TRUE  >
               <TRUE   FALSE  FALSE  FALSE >>.
 -=> ((() () () ()) (() TRUE () TRUE) (TRUE () () ()))

DECLARE BDHEIGHT 3.
 -=>3
  DECLARE BDWIDTH  4.
 -=>4

DEFINE MAZE (START GOAL PATH)
      IF SAMESPOT:<START GOAL> THEN PATH
  ELSE 1:MOVES:<START GOAL PATH>.
 -=>MAZE

DEFINE MOVES (START GOAL PATH)
     [MOVE      MOVE       MOVE       MOVE]:<
    < PATH      PATH       PATH       PATH>
    <START     START      START      START>
    < GOAL      GOAL       GOAL       GOAL>
    <<0 1>    <0 -1>    <1  0>     <-1 0>> >.
 -=>MOVES

DEFINE MOVE (PATH (SX SY) GOAL (MX MY))
     MOVEHELP:<<PLUS:<SX MX> PLUS:<SY MY>> GOAL PATH>.
 -=>MOVE

DEFINE MOVEHELP (START GOAL PATH)
      IF NOT:BLOCKED:START
         THEN MAZE:<START GOAL CONS:<START PATH>>
    ELSE UNKNOWN.
 -=>MOVEHELP

DEFINE BLOCKED  (X Y)
    OR:<LESS:<X 1>  LESS:<Y 1>
        GREAT:<X BDWIDTH>  GREAT:<Y BDHEIGHT>
        X:Y:BOARD >.
 -=>BLOCKED

DEFINE OR LIST
      IF NULL:LIST THEN FALSE
  ELSEIF FIRST:LIST THEN TRUE
    ELSE OR:REST:LIST.
 -=>OR

DEFINE SAMESPOT ((SX SY) (GX GY))
      IF SAME:<SX GX> THEN SAME:<SY GY> ELSE NIL.
 -=>SAMESPOT
```

Figure 2.6-1a

```
          MAZE:<<1 1> <1 1> NIL>.
 -=>()

MAZE:<<3 2> <1 1> NIL>.
 -=> ((1 1) (2 1) (3 1))

MAZE:<<4 3> <1 1> NI
* TIME LIMIT *
T,100

* TIME LIMIT *
T,100
L>.

 -=>-=>-=> MEMORY IS EXHAUSTED.
    -=> YOU HAVE SPECIFIED 7000 NODES,
AND THE LIMIT IS 7000.

-  PROGRAM TERMINATED AT: 000102 IN NEWNODE
```

Figure 2.6-1b

Example 2.7 A Guarded Conditional

In A Discipline of Programming [2], Dijkstra proposes that two nondeterministic programming constructs; a guarded conditional statement and a looping statement, are sufficient for algorithmic control structures. FONS provides an immediate implementation of the first, in which one of a collection of routines is executed if its associated predicate "guard" is true.

The function GCOND (Figure 2.7-1) supplies as many instances of the auxiliary function GUARD as necessary to test all the guards. The collection of calls to GUARD is a multiset, and so successful predicate-value pairs are ordered as they converge. [10].

In addition, this example shows the behavior of the functions AND and OR on multiset arguments. In calls to these functions, predicates are included which yield error messages when fully evaluated, but in each case a value is returned prior to the message because less expensive elements converged first.

```
  ---=>--=>-=> ENTERING VERSION 0.1

 -=> MEMORY LIMIT


  -=> (TRACE 1)

DEFINE AND  LIST
       IF NULL:LIST THEN TRUE
  ELSEIF NOT:FIRST:LIST THEN <>
    ELSE AND:REST:LIST.
 -=>AND

DEFINE OR  LIST
       IF NULL:LIST THEN <>
  ELSEIF NOT:FIRST:LIST THEN OR:REST:LIST
    ELSE FIRST:LIST.
 -=>OR

DEFINE INTEGERS N CONS:<N INTEGERS:ADD1:N>.
 -=>INTEGERS

DEFINE UNDEFIND (X) IF X THEN UNDEFIND:<0> ELSE UNDEFIND:<0>.
 -=>UNDEFIND

DEFINE GCOND LIST 1:[GUARD *]:<LIST>.
 -=>GCOND

DEFINE GUARD  ((P E))
      IF P THEN E
    ELSE UNKNOWN.
 -=>GUARD

AND:[ <PLUS <PLUS PLUS>>:<
      < 2   < 2    2  >>
      < #   < 3    #  >>
      < 4   < 4    4  >> >
      <PLUS PLUS>:<
      < 5    6  >
      < 6   UNB > >          ; NOTE THE UNBOUND LITERAL.
      SAME:<2 3>        ].
 -=>()

OR:[UNKNOWN UNKNOWN NIL TRUE].
 -=>TRUE

GCOND: << 100:INTEGERS:1        "WORST   >
         <  50:INTEGERS:1        "BETTER >
         <  UNDEFIND:<0>         "MIRACLE>
         <  10:INTEGERS:1        "BEST    >
         <   5:INTEGERS:1        UNDEFIND:<0>>>.
 -=>BEST

EXIT.
-=>-=>-=>-=> LEAVING.
NODES DISPOZED, 5763
NODES RECYCLED, 1096
AVAIL--> 580
/
```

Figure 2.7-1

Example 2.8 A Recursive Conditional

Suspensions yield a lesser fixed point semantics for applicative languages than can be found in LISP (Example 2.2), but a further extension of the language solves the problem of undefined predicates in conditional statements [10].

We want the conditional statement:

If P then A else A

to converge to A regardless of the divergence of P, and further, the statement:

If P then A else B

should converge if P is divergent but A and B evaluate to the "same" structure.

The function PARIF in this example uses multisets to achieve this behavior. The predicate and two alternatives are evaluated simultaneously; if the predicate refuses to converge the alternatives are compared and one is returned if they are equal. If the alternatives are both ferns, a copy is made as long as they are element-wise equal, postponing the dependence on the outcome of the predicate.

Two nearly identical runs of this example are shown. In Figure 2.8-1, the last call to PARIF returns the equal part of the alternatives before consuming the rest of its space evaluating the divergent predicate.

In Figure 2.8-2, the call to the function UNDEFINED is replaced by the atom UNKNOWN, a special symbol which stands for a recognizably divergent computation. UNKNOWN causes the program to act as though an evaluation error had occurred. In this case, the printed result is terminated with the error symbol #BOTTOM#, allowing interaction to continue.

```
 ---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT


 -=> (TRACE 1)

DEFINE UNDEFIND (X)  IF X THEN UNDEFIND:<0>  ELSE UNDEFIND:<0>.
 -=>UNDEFIND

DEFINE INTEGERS  N  CONS:<N INTEGERS:ADD1:N>.
 -=>INTEGERS

DEFINE AND  LIST
      IF NULL:LIST THEN TRUE
  ELSEIF NOT:FIRST:LIST THEN NIL
    ELSE AND:REST:LIST    .
 -=>AND

DEFINE PARIF (P T E)
      1:[ SYSIF:<P T E>
          SYSIF:< AND:<ATOM:T ATOM:E SAME:<T E>>  T  UNDEFIND:<0>>
          SYSIF:< AND:<NOT:ATOM:T NOT:ATOM:E>
                  CONS:< PARIF:<P FIRST:T FIRST:E>
                         PARIF:<P REST:T REST:E>    >
                  UNDEFIND:<0> >].
 -=>PARIF

DEFINE SYSIF (PRED THENPART ELSEPART)
      IF PRED THEN THENPART ELSE ELSEPART.
 -=>SYSIF

PARIF:< AND:<1 2 3 NIL> ^BETA ^BETA>.
 -=>BETA

PARIF:< AND:<1 2 3 NIL> ^BETA ^GAMMA>.
 -=>GAMMA

PARIF:< UNDEFIND:<0>
        <<1> 2 3>
        <<1> 2 3>  >.
 -=> ((1) 2 3)

PARIF: <  10:INTEGERS:1
          <1 2 4 6 8 10>
          <1 2 4 6 16 32>  >.
 -=> (1 2 4 6 8 10)

PARIF:<  UNDEFIND:<0>
         <1 1 1 1>:<INTEGERS:1>
         <1 2 3 5>   >.
 -=> (1 2 3

 -=>-=>-=> MEMORY IS EXHAUSTED.
    -=> YOU HAVE SPECIFIED 7000 NODES,
AND THE LIMIT IS 7000.


-  PROGRAM TER
```

Figure 2.8-1

```
 ---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT


 -=> (TRACE 1)

DEFINE INTEGERS  N  CONS:<N INTEGERS:ADD1:N>.
 -=>INTEGERS

DEFINE AND  LIST
      IF NULL:LIST THEN TRUE
  ELSEIF NOT:FIRST:LIST THEN NIL
    ELSE AND:REST:LIST   .
 -=>AND

DEFINE PARIF (P T E)
      1:[ SYSIF:<P T E>
          SYSIF:< AND:<ATOM:T ATOM:E SAME:<T E>>  T  UNKNOWN>
          SYSIF:< AND:<NOT:ATOM:T NOT:ATOM:E>
                  CONS:< PARIF:<P FIRST:T FIRST:E>
                         PARIF:<P REST:T REST:E>   >
                  UNKNOWN >].
 -=>PARIF

DEFINE SYSIF (PRED THENPART ELSEPART)
      IF PRED THEN THENPART ELSE ELSEPART.
 -=>SYSIF

PARIF:< AND:<1 2 3 NIL> ^BETA ^BETA>.
 -=>BETA

PARIF:< AND:<1 2 3 NIL> ^BETA ^GAMMA>.
 -=>GAMMA

PARIF:< UNKNOWN
        <<1> 2 3>
        <<1> 2 3>  >.
 -=> ((1) 2 3)

PARIF: <  10:INTEGERS:1
          <1 2 4 6 8 10>
          <1 2 4 6 16 32>  >.
 -=> (1 2 4 6 8 10)

PARIF:<  UNKNOWN
         <1 1 1 1>:<INTEGERS:1>
         <1 2 3 5>   >.
 -=> (1 2 3 #BOTTOM#)

EXIT.
-=>-=>-=>-=> LEAVING.
NODES DISPOZED, 19735
NODES RECYCLED, 2873
AVAIL--> 533
/
```

Figure 2.8-2

Chapter 3. Program Notes

Section 3.1 Introduction

These program notes give an overall look at the interpreter as a machine independent program. We describe the fundamental data structures and behavior here, without regard to the implementation language or host machine, concentrating on basic design variants. As the program is both a vehicle for applicative programming and a computational model, there are times when issues of efficiency are ignored, in favor of generality. This is most obvious in the section on evaluation. A syntax for the interpreter's language has not been fully established; modeling strategies are in the beginning stages of development. Discussion of these aspects are open-ended, to be dealt with more fully in Chapter 7.

The program is divided into four modules, responsible for memory management, evaluation, input, and output. A subsection is devoted to each module, and there are additional discussions of the basic storage constituants, universal structures and operations, and the Fern evaluation strategy. Chapter 6, which contains implementation specifics, has a more detailed section for each section here. Throughout the program notes, the following conventions have been adopted for special words:

a. Interpreter function names and program variables are upper case;
   FIRST, ASSOC, LIST . . .

b. Data types begin with an upper case character;
   Pname, Suspension . . .

c. Data fields are lower case, underlined;
   ref, pname, car.numberp . . .

d. Procedure names are upper case and underlined;
   DISPOZE, EVAL

Section 3.2 Cells

The interpreter's memory is a collection of cells (or words, or nodes) which are of uniform size. Cells may be of type Fern, Atom, or Suspension, and each of these types is subdivided further as discussed below. With the exception of certain Suspensions, all cells have three pointer fields called ref, car and cdr; with each pointer field there is a flag , numberp, which states explicitly whether the content of that field is to be interpreted as a pointer or as an integer (data). These and three more flags are used by the interpreter for cell type identification. In all, then, there are six flags and three fields in most cells:

1. atomp -- cell-typing flag
2. pname -- cell-typing flag
3. multi -- cell-typing flag
4. ref -- pointer/integer field
5. ref.numberp -- field specification flag
6. car -- pointer integer field
7. car.numberp -- field specification flag
8. cdr -- pointer/integer field
9. cdr.numberp -- field specification flag

The Type Suspension

Suspensions are system structures and cannot be accessed by the user. There are two kinds, Pnames and Free- (or Linked-) stacks. Pnames are highly specialized single-cell structures which occur in highly specific contexts. Because of this, the term Suspension is used to denote non-Pname Suspensions. Pnames contain the character codes for literal atoms instead of the pointer fields mentioned above. This is the only cell type without pointer fields. Suspensions, in the more specific denotation, have pointers in their ref and cdr fields; the car field may contain either a number or a pointer. The cdr of a Suspension is either

NIL or another Suspension. The pname flag is the type specifier for Pnames. In Suspensions, the pname, atomp, and multi flags are usually false; the type determining characteristic is the existence of a pointer in ref (ref.numberp = TRUE).

The Type Atom

Atoms may be either Literal or Numeric. All atoms have a number in their ref field, and TRUE in their atomp flag. Numerics have a number in car (their value) and do not use the cdr. In Literals, the car and cdr are pointers: cdr to the Atom's Pname, car to the next Atom in a hash bucket.

The Type Fern

Fern cells have pointers in their car and cdr fields, a number in their ref fields. FALSE in the atomp flag specifies Ferns as non-Atomic. The two subtypes are List and Multiset, differentiated by the multi flag. The pointer fields may refer to Atoms, Ferns, or Suspensions, but not to Pnames.

The ref Field

When the ref field of a cell is a number it tabulates the references to that cell in the system, a reference count. When a pointer assignment is made, the reference count of the object is incremented. Instead of using a garbage collector, the interpreter has a cell recycler, sensitive to reference counts, which reclaims cells when they are no longer referenced. Atoms and Ferns, that is, user structures, have reference counts; Pnames and Suspensions do not. Therefore, system structures must be uniquely referenced. In short, all Suspensions and Pnames have a reference count of one.

Graphic Representation

Figure 3.2-1 depicts the Fern structure: (EQUIV (a 5)). In the List representation, the lower case character strings are Suspensions which have not yet converged. We adopt the convention that the Suspension a will evaluate to the value A, so that fully coerced, this Fern is (EQUIV (A 5)). In the graphic representation, cells are denoted as boxes; the way a box is divided indicates the type of the cell. The cells C1, C2, C4, and C5 are type Fern, divided into three fields, the leftmost field containing the reference count. The mark at the upper-left of C4 indicates that it is type Multiset. C3 and C6 in the figure are type Atom, the Literal EQUIV and the Numeric 5 respectively. C3's associated Pname is excluded. Cell C7 is a Suspension, divided horizontally, so that it resembles a stack. The lower case form a in the cell shows that C7 will converge to A. The ground symbol is NIL.

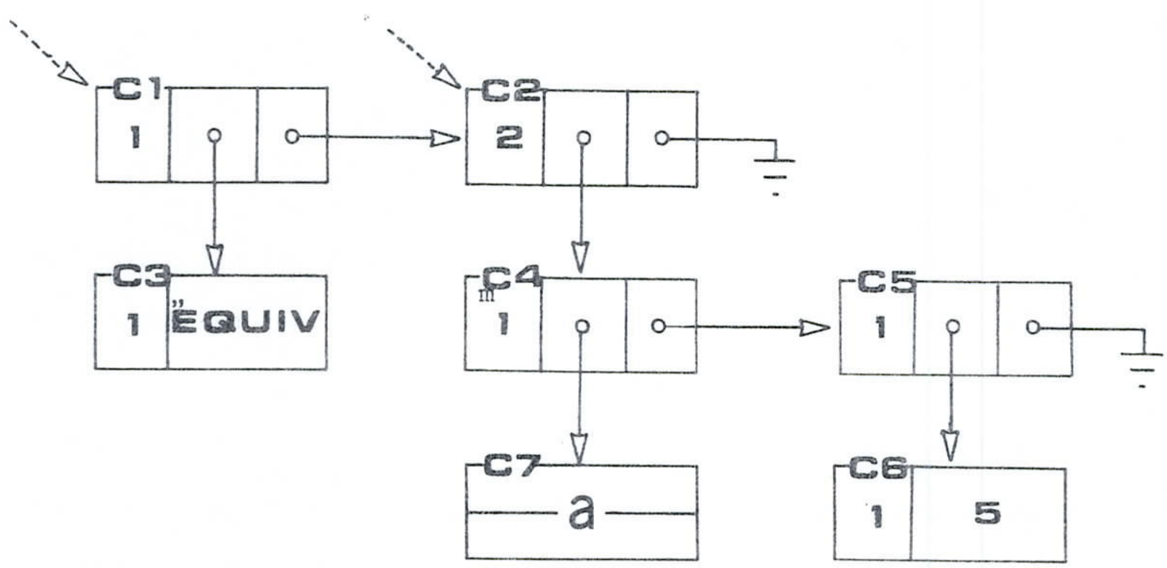


FIGURE 3.2-1

Table 3.2-2 summarizes cell typing:

Type / Subtype	ref	car	cdr	General Comments / comments
1. Suspension				1. Uniquely referenced; no user access.
a. Pname	------------character code------------			a. pname is TRUE.
b. Suspension	pointer	pointer or number	pointer	b. cdr is either NIL or a Suspension.
2. Atom				2. atomp is TRUE.
a. Literal	reference count	pointer	pointer	a. cdr is Pname; car is next bucket entry.
b. Numeric	reference count	number	unused	b. car is value.
3. Fern				3. Pointers may be to Ferns, Atoms, Suspensions
a. List	reference count	pointer	pointer	a. multi is FALSE.
b. Multiset	reference count	pointer	pointer	b. multi is TRUE.

Table 3.2-2

Section 3.3 Stacks and Assignment

The most pervasive structure in the interpreter's system is the stack; stacks are used in all of the program modules. The term "stack" is used to describe a conceptual entity, an abstract structure type, and not to connote implementation features. Specifically, the system contains no sequential stack-like structures: stacks are in fact linked, and free floating in memory, competing with user structures for available space.

As an abstract data type, stacks admit four operations. The equality predicate is used to test for the empty stack, NIL; the assignment statement is used to inspect the topmost stack entry; and procedures are written to do the elementary operations PUSH and POP. The restriction of inspection to the topmost entry is strict, to guarantee that stack elements never have multiple references. This leaves the ref field available for information.

The choice of which field to use for stack linkage varies, although it is usually the cdr field. Each module, therefore, has its own set of stack operators, tailored to use a particular link. Variables local to the modules but global to the operators are used to specify particular stacks. Each stack entry contains two items of information along with the stack link. Additional information is sometimes placed in the type flags, when the context makes it possible.

One of the more arduous tasks of the system programmer is to keep track of all references, and to maintain reference counts. This must be totally transparent to the user. Local variables are especially bothersome because they are not always legitimate references. Trailer and pointer variables are frequently used to simplify algorithms, but gains in speed and clarity are nullified if every assignment involves an adjustment in reference counts. On the other hand, occasions when a local variable constitutes the only reference must be tabulated if the cell is to be recycled at the right time.

To solve these problems, pointer variables are divided into two categories, each with its own assignment operator. When the program is viewed as a machine model, these variables play the role of registers, and are called inspection-registers and value-registers. Inspection-registers contain very temporary cell references used by the program to extract decision making information from cells. If the program is to do one thing if the FIRST:FIRST:REST:cell is Atomic, and something else if it is a Suspension, inspection-registers can be used to probe the structure without the cost of changing reference counts. Trailer variables are usually in the inspection category.

Value-registers are used to hold information. Because their contents are valid between procedures,the assignment operator for value-registers adjusts reference counts. It releases the previous contents of the variable and increments the count of the new contents:

```
VALUE-REGISTER-ASSIGNMENT (register, value)
         begin
         INCREMENT-REFERENCE COUNT(value);
         RECYCLE(register);
         register := value
         end
```

In terms of memory access, Value-register assignment is a costly operation. Its expected speed is improved if a test is included to see if the register's contents need to be changed.

Equally expensive are field assignments, which also require reference count adjustments. A cell is effectively a set of three value-registers. Procedures for car, cdr and ref assignment are like VALUE-REGISTER-ASSIGNMENT, but in most cases the programmer will prefer to express them in-line, in the code, subtracting the cost of parameter binding.

Section 3.4 Memory Management

Memory is an array of cells bounded by zero and the constant MEMORYSIZE. MEMORY[0] represents NIL, and cannot be examined by the user. Having a cell for NIL simplifies the control structure of traversal algorithms and may be of use to a processor model. The segment from location 1 to OBLISTSIZE is reserved for a hash table. The remainder is called free space and contains all system and user structures, including stacks. All non-active cells are elements of a global stack called AVAIL. Figure 3.4-1 shows the initial memory configuration.

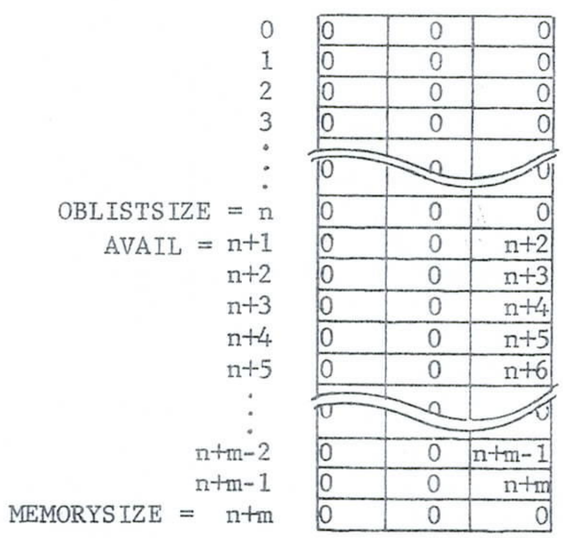


Figure 3.4-1

The program starts with an initialization phase. At this time, field contents are cleared and the AVAIL stack is established by linking free space sequentially through cdrs. All other fields point to NIL; all cells are type Fern. Next, system Atoms are declared, and a top level environment is constructed. This consumes some of free space, and ends the initialization phase.

Four procedures form something of a kernal for a larger set of structure manipulators. They are NEWNODE, NUDGE, DISPOZE*and RECYCLE. Beyond this kernal are routines to do assignment, hashing, Fern construction, and stack manipulation. A few predicates are provided to do cell inspection (Figure 3.4-2).

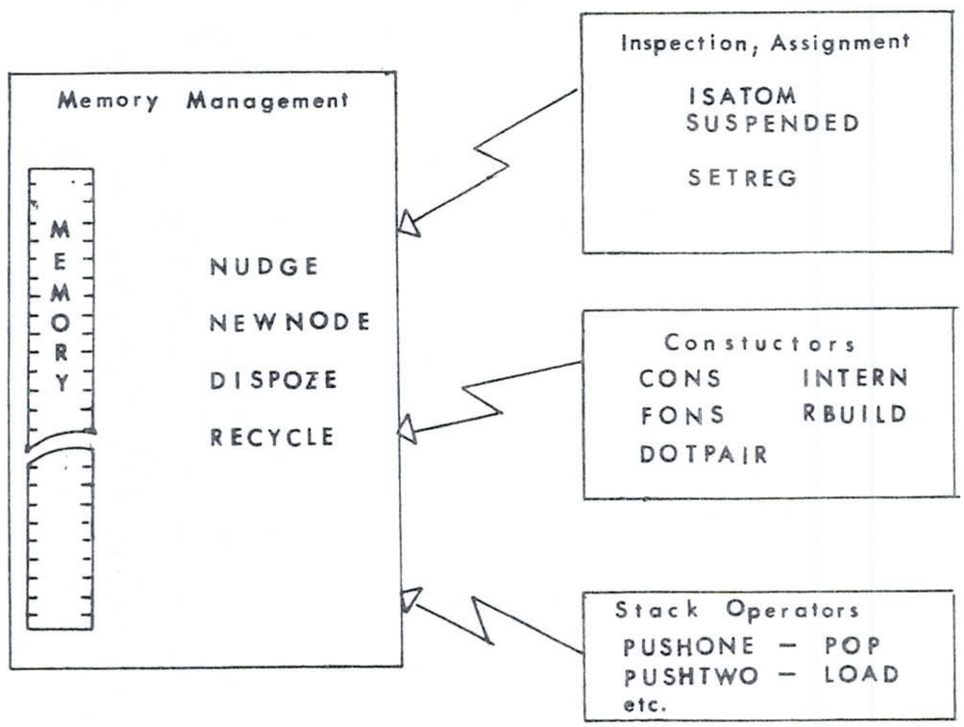


Figure 3.4-2

*To distinguish it from PASCAL's procedure DISPOSE.

It is not uncommon to bypass the management routines in order to do straightforward operations, like field assignments. We describe the four management elements individually here.

The procedure NUDGE takes one argument, a pointer. The corresponding cell is inspected, and if it is NIL or a Suspension, nothing is done. (Suspensions are NUDGEd only when doing so simplifies the code.) If the cell is an Atom or a Fern, its reference count is incremented.

The procedure DISPOZE does a push on the AVAIL stack. NIL and Literal Atoms are ignored; Literals are recovered during hashing. The contents of the cell are erased, or equivalently, the information pushed onto the stack consists of two pointers to NIL. DISPOZE does not look at reference counts, and is called only when the cell involved is known to be unreferenced. Stack pops, for example, will load the information in the topmost cell into value-registers and then DISPOZE that cell.

RECYCLE is used to reclaim structures. Its single argument is a pointer and if the corresponding cell has a reference count larger than one, it is decremented and nothing more is done. Atoms are DISPOZEd; with other structures, cdr links are followed until NIL, an Atom, or a reference count larger than one is discovered. The resulting List is appended to AVAIL.[1] The ref and car fields of these cells are not erased as in DISPOZE. Some discussion of this reclamation strategy is described by Knuth, [11, Section 2.3.5]. Its primary advantage is that it requires only one visit per cell returned, thus making up some of the cost of assignment.

---

[1]This recycling algorithm is linear with respect to the List it is returning, which violates the requirement that the reclamation primitive be O(1). But see the note in Section 7.7.

NEWNODE is an AVAIL pop. It fetches the top cell from the stack and RECYCLEs its car and ref pointers. DISPOZE erases its cell to avoid trouble here. NEWNODE takes no arguments and returns a pointer to the new cell.

After establishing AVAIL system Atoms are declared and a top level environoment is created. The function SYSATM is given a character array, which it HASHes into an OBLIST bucket. NEWNODE is called to provide free cells and a Literal is created. They are given a reference count of two so that it can't be DISPOZEd. The structure of the OBLIST is depicted in Figure 3.4-3.

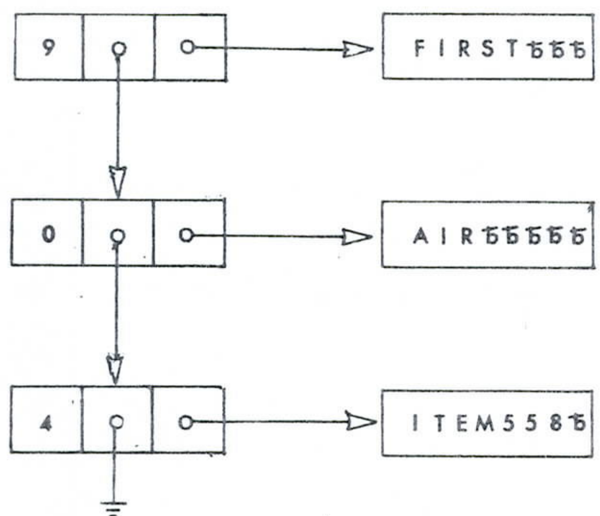


Figure 3.4-3
A Hash Bucket.

The top level environment has two parts, an association table for constants -- initially binding TRUE to TRUE -- and one for functions -- initially binding LIS to (LAMBDA LIS LIS) , the LISP-like LIST primitive.

Section 3.5 Evaluation

Structure

EVAL has two kinds of procedures, service routines to do local stack manipulation and so forth, and eval-procedures which implement the semantics of the language. Eval-procedures are labeled sections of code which receive their arguments globally through stack pops and call their peers through pushes. This interaction is driven by a loop:

```
                LOOP

repeat
  1.  Pop the EVAL STACK; the
      result is an eval-procedure
      name and an argument.
  2.  Branch to the specified
      eval-procedure.
until the STACK is empty
```

This loop is not outermost with respect to evaluation, however. There are actually two stacks involved; the second one, STAQUE, contains a sequence of contexts consisting of a cell address, a Suspension, and a field flag:

```
EVALUATE:

while STAQUE is not empty do
   begin
     1.  Pop STAQUE to get a
         new context.
     2.  Place the current result
         in the context's cell,
         according to the context's
         flag.
     3.  Set STACK to the context's
         Suspension.
     4.  LOOP.
   end
```

As an example, suppose we evaluate FIRST: C1.

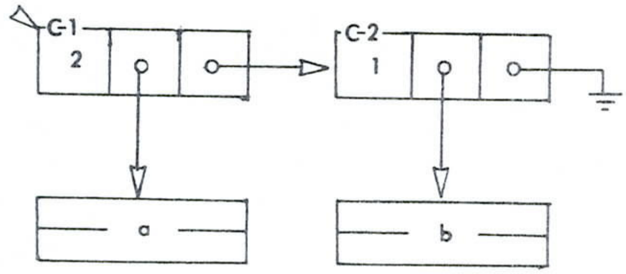


Figure 3.5-1

Upon entry, EVAL gets a value for STACK:

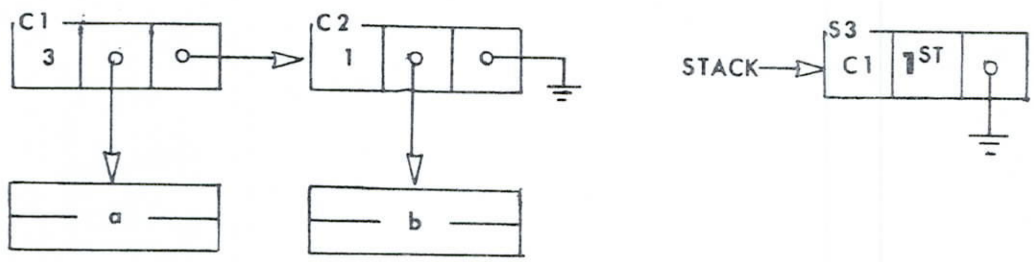


Figure 3.5-2

The eval-procedure FIRST examines cell C1 and finds a Suspension. The old STACK, S3, is pushed onto STAQUE and the Suspension a becomes the current STACK.

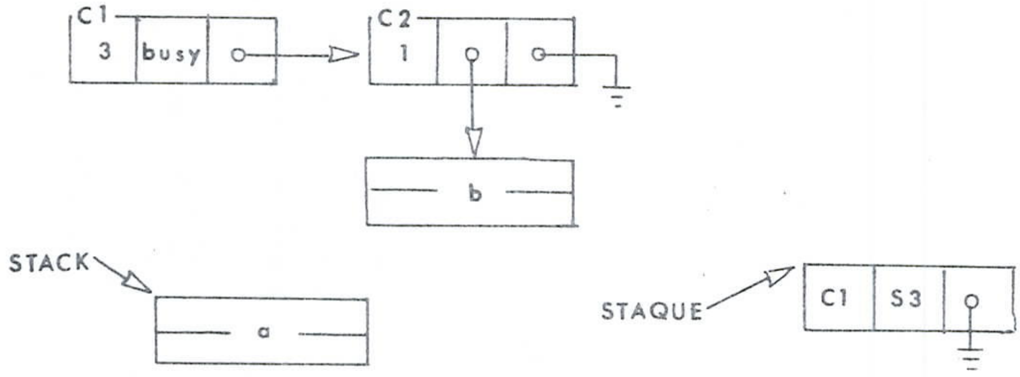


Figure 3.5-3

LOOP procedes on the Suspension a, let us say, until it is exhausted, yielding the result, A. Since STACK is NIL,STAQUE is popped; the context specifies that the result goes into the car of C1:

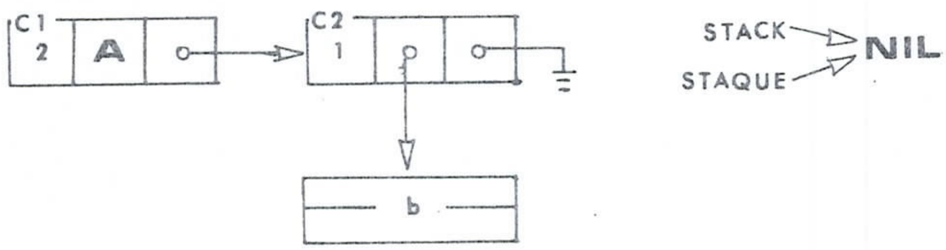


Figure 3.5-4

Now both STACK and STAQUE are empty, so the result is returned as a value by EVAL.

EVAL uses ten local registers (see section 3.3), including STACK and STAQUE. The stack pointers are inspection type, since it is invariant that stacks be uniquely referenced. Three additional inspection-registers are used for privileged structure examination. Their values are not valid between eval-procedure calls. The other five variables are value-registers. One, (ENVDOT) is reserved, and points to the current Environment. Another, (REVAL), is used to pass values among eval-procedures. The remaining value-registers are argument pointers for eval-procedures.

STACK points to active Suspensions. To make the program compatible with multiprocessing, when STACK is assigned, the corresponding cell field is "invalidated", (Figure 3.5-3) so that concurrent evaluators can't access the same Suspension. These invalid fields contain <u>resource allocations</u> for the evaluator. Every call to <u>FIRST</u> or <u>REST</u> may involve a context change and a distribution of resources: some for the old context, the rest for the new one. It is possible that in a given context, resources are exhausted before a value is found. When this happens, a STAQUE pop is forced and the result returned to the cell is a new Suspension, not a value.

Resources are a recent addition to the model, required to enable Multiset evaluation. The strategy for allocation is straightforward (half for each context) and probably too simplistic. Other proposals are presented in Chapter 4. The invalid cell field is the appropriate place to hold the resource number; competing processes could adjust each other's behavior by making changes in the field.

Entries on Suspensions have two formats: car contains either a number or a pointer. When the field contains a number it is an encoded Eval-procedure name; when it is a pointer it is an argument. Eval-procedures require from two to five arguments, including REVAL and ENVDOT, so from zero to three arguments are held on STACK. The specific format of a STACK element is always context dependent -- both the calling and called Eval-procedures know the order and number of the arguments -- and no confusion results from having two types of entries.

Behavior

All system-defined functions are Eval-procedures. Because of the size of the model, arithmetic and logical primitives are few in number. The semantic behavior of EVAL is the classic EVAL-APPLY figure-eight loop, except that it is interrupted by context changes. A more graphic description would be a tower of such loops, each interrupted by the one above.

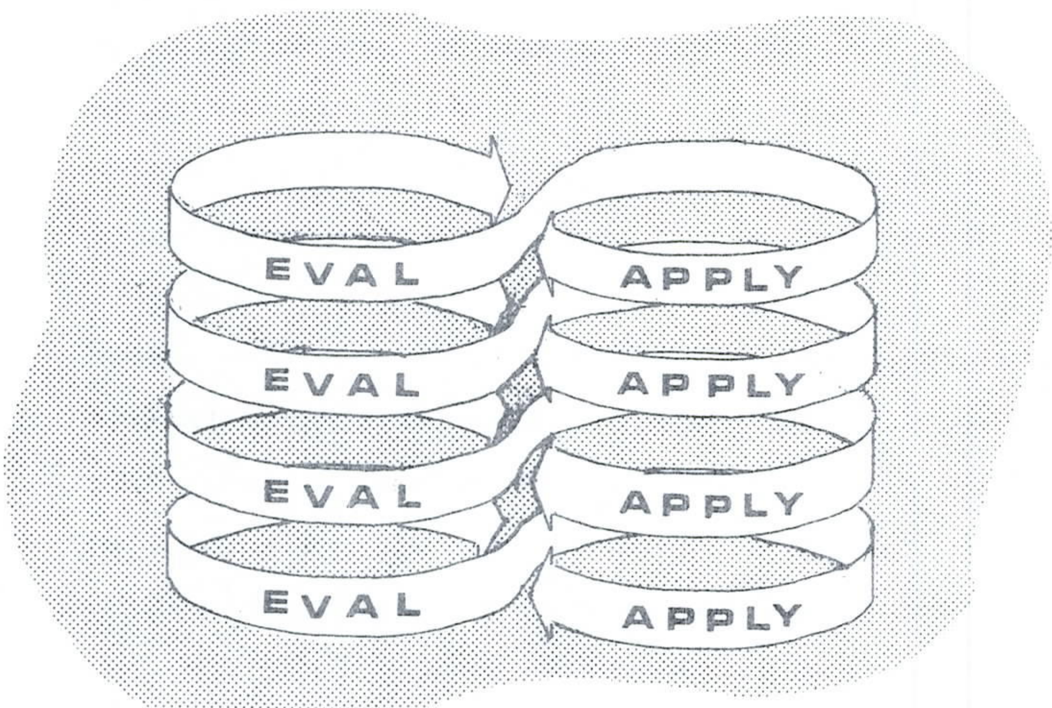


Figure 3.5-5

When a structure probe uncovers a Suspension the current loop stops and adds a new loop to the top of the tower.

Nearly all functions evaluate their arguments, but some exceptions are allowed deep within the system. The purpose of these exceptions is to allow indirect access to Suspensions. Suspensions themselves cannot be shared but the cells which contain them can; _special forms_ are used to do this, so that extraneous _access environments_ can be avoided. For example, the CONS operator might behave like this:

```
. . . In the EVAL part of the loop,

  if the function name is CONS then
     1.  EVALuate the argument.
     2.  APPLY CONS to the result.

. . . In the APPLY part of the loop,
  if the function is CONS then
     1.  Create a new environment
         binding #ARG to the argument.
     2.  Build the form FIRST:#ARG
     3.  Build the form FIRST:REST:#ARG
     4.  Construct a suspended List
         with the two forms in the
         new environment.
```

A probe on the resulting List causes a context push on STAQUE. Evaluation of either form requires that #ARG's binding be found in the List's environment. The construction of this environment and the subsequent searches are avoided by establishing two system functions which access the proper Suspensions immediately:

```
. . . In the EVAL part of the loop,
      if the function name is CONS then
         begin
         1.  EVALuate the argument.
         2.  APPLY CONS to the result.
         end

. . . In the APPLY part of the loop,
      if the function is CONS then
         begin
         1.  Build the form #FIRST:argument.
         2.  Build the form #FIRSTREST:argument.
         3.  Construct a List with no environment
             pointing to the suspended forms.
```

When this List is probed and the Suspension evaluated, the form #FIRST: argument is intercepted by EVAL; no argument evaluation takes place. The value returned is immediate (unless it too is suspended) and no call to APPLY is made.

Relatively speaking, the EVAL half of the figure-eight is more passive than in LIST interpreters: (see Figure 3.5-6)

In summary, evaluation is driven by two stacks, one pointing to a sequence of contexts, the other to a Suspension. Suspension stack elements contain pointers to a set of Eval-procedures, which communicate through the Suspension and by assignment to value-registers. Structure probes may uncover new Suspensions, causing context pushes. The evaluation of a Suspension does not guarantee that it will be reduced to a result; it may instead be replaced by a more advanced suspension. The time spent on a given Suspension is determined by the distribution of resources during context change.

```
EVAL (form, environment)

     if the form is NIL or a Numeric
         then return it
     else if the form is Atomic
         then start an environment search.
     else if the FIRST:form is a special
             system function (#FIRST, etc.)
         then execute it.
     else
         begin
         1.  EVALUATE the argument in environment.
         2.  APPLY the function to the result.
         end


APPLY (function, argument, environment)

     if the function is system-defined
         then coerce the proper argument fields.
          and call the proper Eval-procedure.
     else if the function is user-defined
         then
         begin
         1.  EVAL the function in the environment
         2.  APPLY the result to the argument.
         end


     else if the function is a LAMBDA form
         then
         begin
         1.  Bind the formal parameter to the
             argument.
         2.  EVAL the function body in the
             new environment
         end


     else
         begin
         1.  APPLY FIRST:form to the first column of the
             argument.
         2.  APPLY REST:form to what is left of the argument.
         3.  return the CONS/FONS of steps one and two.
         end
```

Figure 3.5-6

Section 3.6 Output and Input

Output is the reduction of a fern to a character string, or equivalently, a trace of an in-order structure traversal. Given a primitive, WRITE, which expands print names, the algorithm for the reduction has a common recursive form:

```
DEFINE PRINT STRUCTURE
        if NULL:STRUCTURE then WRITE:NIL
    elseif ATOM:STRUCTURE then WRITE:STRUCTURE
      else WRITE:"(
           then PRINT:FIRST:STRUCTURE
           then AUXPRINT:REST:STRUCTURE
           then WRITE:").

DEFINE AUXPRINT STRUCTURE
        if NULL:STRUCTURE then DONOTHING
    elseif ATOM:STRUCTURE then WRITE:".
                          then WRITE:STRUCTURE
      else PRINT:FIRST:STRUCTURE
           then AUXPRINT:REST:STRUCTURE.
```

(N.B. The iterative construct, _then_ . . . ; _then_ . . . ; _then_ . . . ; is not a feature of the implementation language.) This definition fails to reflect the important aspects of the output routine, however. Its role is that of primary mover in the system, specifying precisely which suspensions are to be coerced and for how long, determining which structures can be recycled into free space. Moreover, a recursive PRINT algorithm consumes stack space needlessly.

Friedman and Wise [7] present an algorithm for _PRINT_ requiring constant space. The argument structure is threaded during traversal and replaces the recursive stack. Their report demonstrates that the printed structure can be returned to AVAIL concurrently during the traversal. This feature, along with a suspending constructor enables constant-space costs in some forms of recursion. With minor changes, their algorithm is used in the model. As a system module, PRINT maintains value- and inspection-registers just like the evaluator. In the report, every variable assignment is accompanied by code which inspects reference counts, DISPOZing cells whose counts are zero. This operation is here subsumed by the assignment operator if these variables are typed as value registers. Nevertheless, PRINT must be sensitive to references at the point that a decision is made whether to include a fern cell in the threaded structure. Externally referenced cells are included; cells referenced only by PRINT are thrown away. Because the recursive version of PRINT does no threading, recycling happens automatically; the space inefficiency is more attractive because no explicit mechanism for reference inspection is needed.

Input is the inverse of PRINT; a sequence of characters is transformed into a data structure for evaluation. The result of the transformation is either an atom or a binary list; when it is a list, the first element is one of a number of special atoms unavailable to the user. For example, the string

FIRST:LIST

is parsed into the list form (APPLY FIRST LIST), but APPLY is actually written ##:##, an atom which cannot be built by the user.

A number of reserved symbols are trapped by the input scanner. A semicolon signifies that the rest of the line is a comment. The backspace and ESCape characters serve their normal functions. A double slash, "//", signals the input routine to reject the form it is building and start over.

The host system automatically buffers the input file, so READ is forced to be line oriented. Evaluation is modeled to take place whenever a full form is constructed, but it actually happens after the next carriage return. Neither of these schemes is satisfactory; the read transformation should be suspended (See Section 7.5).

Section 3.7 Multiset Evaluation

A multiset is an unordered collection of data. Intuitively, these collections resemble sets, but they differ from the mathematical concept in that their elements can be duplicated. Multisets have been added to the system to provide a way to model nondeterministic program behavior (see Examples 2.6, 2.7, and 2.8) and real time computation. A thorough discussion of the semantics of these structures is found in [10].

While multisets are unordered conceptually, manipulating such collections (printing them, for example) necessitates that an order be imposed on them at some point during computation. We shall impose a further constraint: once a first element is chosen, this order is not allowed to change. This requirement preserves transparency.

We state as axioms two properties of multisets as regards the probes.

Axiom 1. If a multiset F contains a convergent element, then FIRST:F converges.

Axiom 2. FIRST and REST are consistent, that is if REST is called on a multiset, thus necessitating a choice for F's first element, a subsequent call to FIRST on F selects the same first element.

FONS and CONS differ only in that the fern cell they create has a different mark in its multi field: TRUE for FONS, FALSE for CONS. The flag determines the evaluation strategy used when probing the fern. When CONS is used to build the structure a probe on the result concentrates on a single field.

We now choose an evaluation strategy for multisets which satisfies the axioms. In making our choice, we are free to impose additional constraints, to satisfy implementation goals, as long as they are not contradictory. We begin by dismissing some pathologies.

One strategy is to impose a multiset order randomly, whenever it's convenient (i.e. at construction time). This is unsatisfactory for at least two reasons. Since construction is suspended, it is not likely that the extent of the multiset will be known at the chosen time. More importantly, the selected first element may be divergent. Divergent candidates must be allowed to drift harmlessly out of reach when possible, to remove themselves from consideration, to satisfy Axiom 1.

A more promising choice is to create a copy of a multiset structure whenever a new element is added, and to apply a sorting algorithm to the structure whenever it is probed. A suspension is selected to be the first element, evaluated for a while, and if it returns a result, it remains in the first cell of the multiset. If the selected suspension consumes its resource allocation without returning a result, another suspension is given a chance. This strategy is rejected, not because of the cost of copying (which is necessary to preserve referential transparency, as shown in the example below), but because the repeated application of the sort is too expensive. The existence of suspensions makes it possible, conceptually at least, to evaluate all of the multiset elements at the same time.

Consider a multiset X with four suspended elements (Figure 3.7-1).

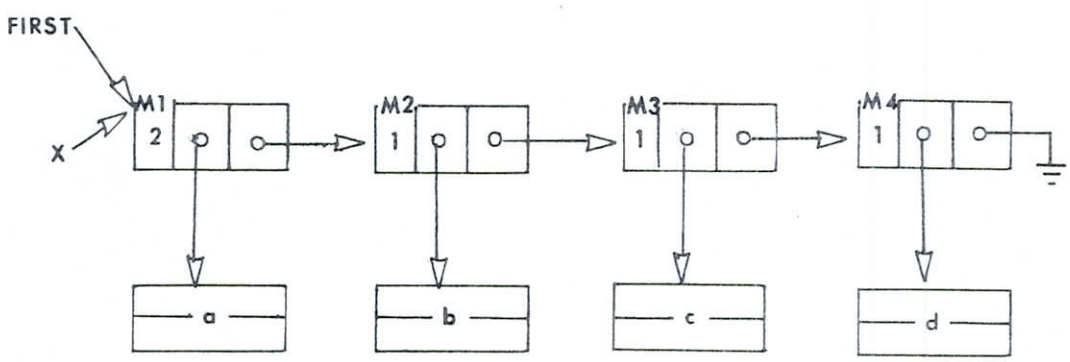


Figure 3.7-1

When FIRST is applied to M1, the Suspensions a, b, c, and d are evaluated simultaneously: whichever converges fastest becomes the first element of X. Suppose c and d converge and the others do not. One of the values, say C, is moved to the front of the multiset. The orphaned suspension, a, might move to the vacated cell, M3. Subsequent calls to FIRST with X would cause no more evaluation; C is returned. (Figure 3.7-2).

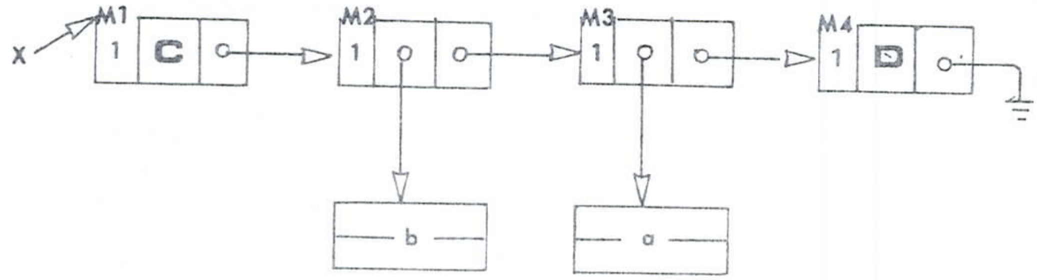


Figure 3.7-2

A problem arises, however, when there are additional references to the interior of a multiset. Let Y = [b c d] and X = FONS:⟨a Y⟩ . For the purpose of this example, suppose the second argument to FONS is coerced. X is a structure much like the one in Figure 3.7-1, except that cell M2 now has an external reference.

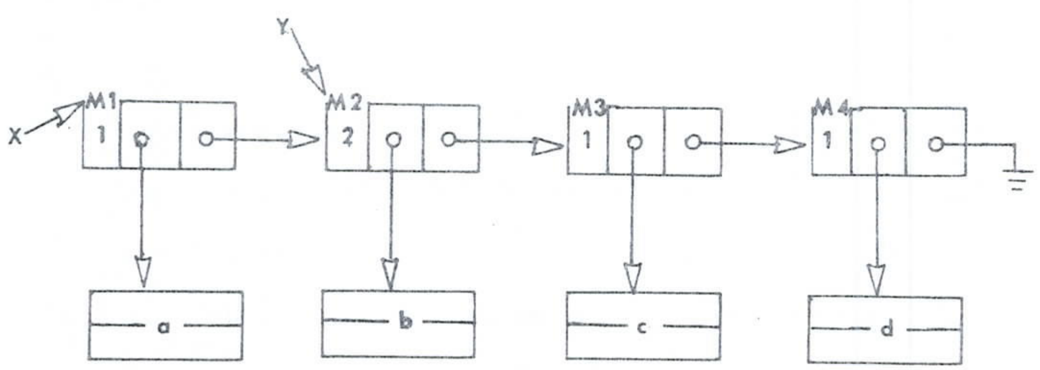


Figure 3.7-3

Now if FIRST is applied to M1, and if the above sorting strategy is used, Y's integrity has been violated:

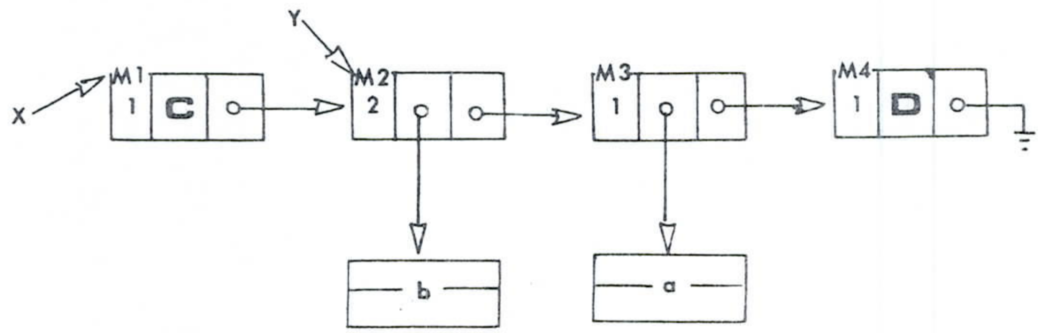


Figure 3.7-4

X now points to [C b a D]; Y to [b a D], and we have violated the transparency condition.

To solve this problem, we could do a full structure copy with each call to FONS, as we mentioned above, but we don't need to waste that much space. Only the cells which are externally referenced need to be copied, and they can be determined by examination of reference counts. Furthermore, this copying takes place at evaluation time, not during construction, so the structure transformation is dynamic. From the point of convergence, the FIRST-value is moved into each of the multiply referenced cells, and a new cell is obtained to hold their suspensions. In compensation for the cost in space, future probes on the internal structures will not have to look for a convergent element. (See Figure 3.7-5).

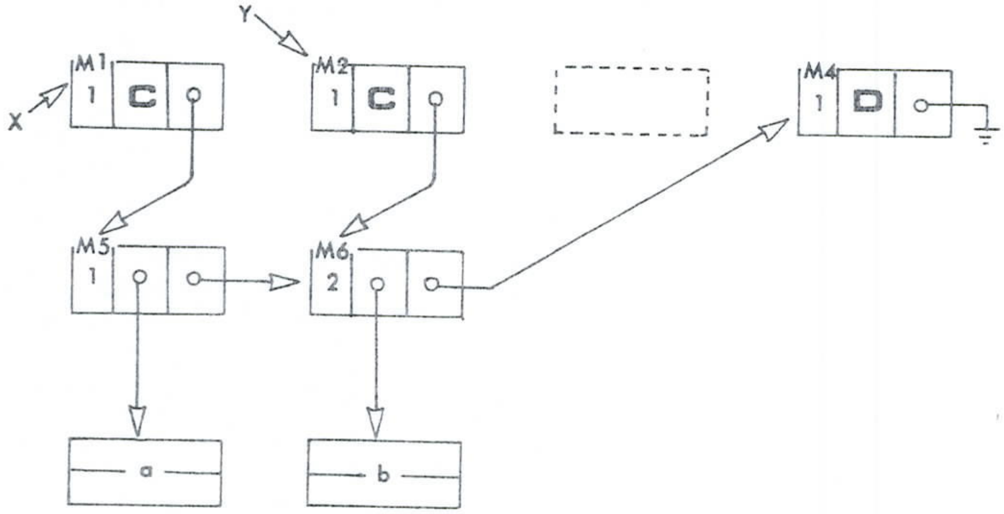


Figure 3.7-5

Figures 3.7-6 and 3.7-7 demonstrate more clearly that only externally referenced cells are submitted to the copying operation. Cells referenced only by their multiset predecessors are moved to the "second level" of the transformation. Figure 3.7-6 is the multiset X = [a b c d e f].

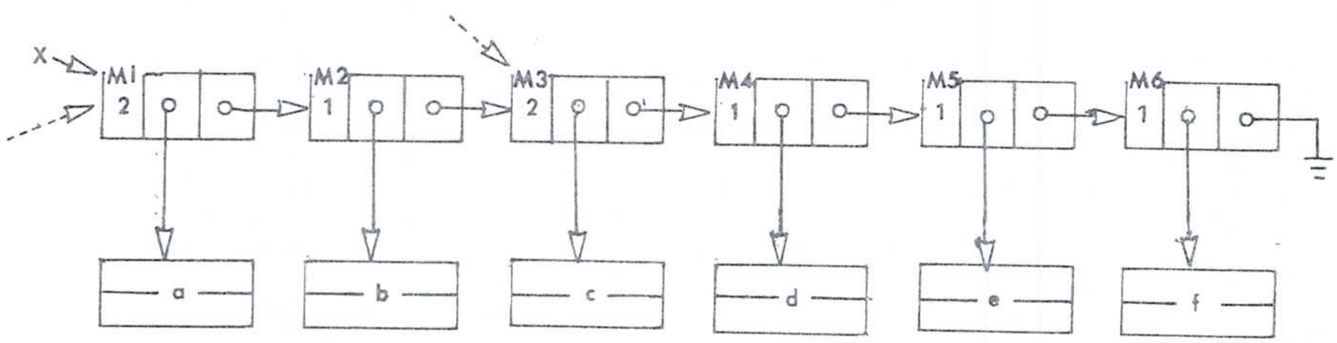


Figure 3.7-6

A probe on X causes the suspension E to converge.

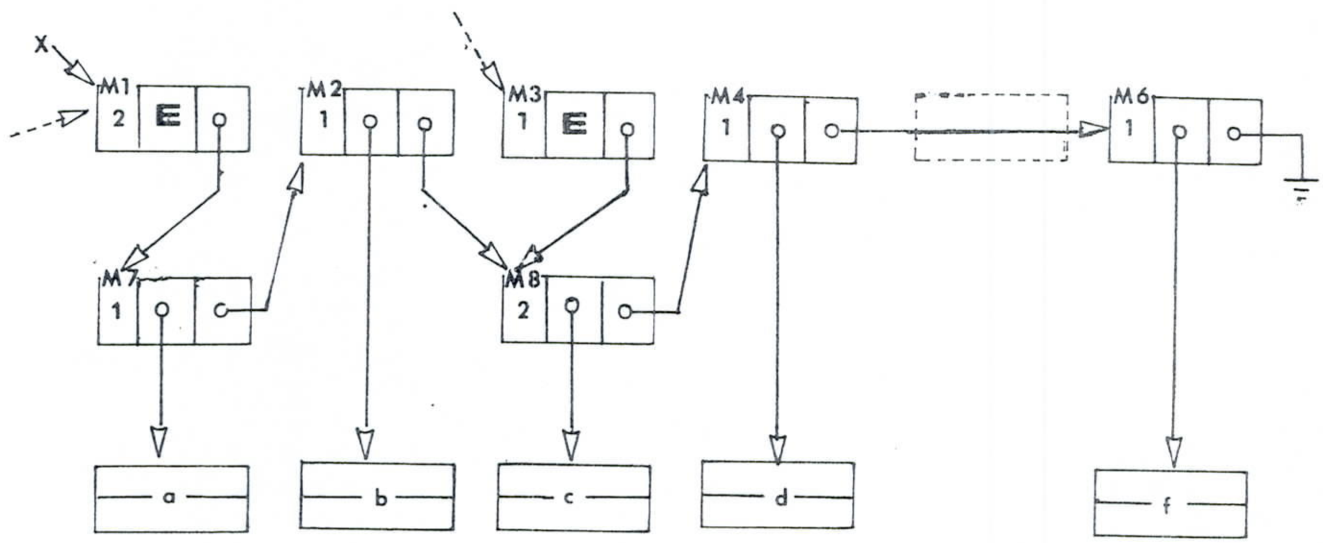


Figure 3.7-7

Finally we point out that because it is precisely the externally referenced cells which are effected by probing, simultaneous fern-wise transformations are prohibited. In a single processor model this means that the transformation code cannot be suspended unless a provision is made to lock other probes out of the transformed structure. A discussion of this constraint can be found in Chapter 7.

Chapter 4. Multiprocessing with the Model

Technological advances in electronics make possible the collection of large numbers of individual processors, expected to act in unison, contributing to the execution of a single algorithm. Such collections already exist in the form of vectored processors, which automate a common programming construct, the finite loop, and reduce it to a single instruction. The result is not merely a speedup in computation, but also a simplification of software. Very Large Scale Integration of circuitry (VLSI), opens the way to construction of computers with an even greater multiplicity of processors. The CPU was once a primary expense in a computer system; it is now a triviality. The cost of computation depends more and more on the maintenance of software, and with respect to multiprocessing, this problem may be approached in a number of ways.

Well defined problems lead to the design of specialized machines for their solutions. This is the impetus behind vectorization; matrix addition is a common programming task, greatly speeded by parallelism. One problem with vectored processors is that they are not specific enough; they straddle the gap between IC component technology and VLSI. Operand size is variable and it is the responsibility of the programmer to establish argument boundaries. If these kinds of parameters are known beforehand, a problem specific machine can be built to manipulate data structures of exactly one size. As problems grow larger and machines cheaper, the demand increases for problem specific hardware. When problem specification determines design, little software is needed.

At the other extreme are general purpose computers which are more flexible albeit slower. There is growing suspicion in the industry that general purpose multiprocessors will not be developed much beyond the current level of architecture, where a few interdependent CPUs, each fortified with adequate defenses against the others, crunch away within well defined boundries of time and space. The reason for this pessimism is again, software. Downward compatibility and communication protocol impose great burdens of the system and the programmer. Expanding current architecture to permit even a few dozen processors increases this burden profoundly.

To give the general purpose multiprocessor just consideration, programming languages must be developed to reflect multiplicity naturally. In turn, the semantics of these languages gives direction to design. One purpose of the model described in this thesis is to take a step in that direction.

Parallelism and Concurrency

There is a need to distinguish between two kinds of simultaneous computation. In a vectored operation, an array of identical arithmetic units perform the same tasks on a set of argument arrays. One can imagine that they fetch their instructions from the same wires, that they are driven by the same clock, and that they present their results at the same instant. This lock-step behavior is called parallelism. A more general term, concurrency, describes a collection of processors contributing to the execution of an algorithm; the members need not be identical or even similar; they are not necessarily executing the same instruction or even the same program. Concurrency might be used to describe a process oriented operating system, but we wish to carry the interpretation farther, to minimize or eliminate the role of a centralized organizing process.

Critical processes, a scheduler for example, should be fool-proof as possible, since the entire system depends on them. Unlike software constructs, where critical code is optimized to its fullest extent, simplicity is an imperative in critical hardware.

Transparency

We require of our architecture that the execution of an algorithm does not depend on the number of processors available for the task. The problems of timing and interaction should be transparent to the user; processor management should be implicit in the software. A number of features of our model have this property. The absence of an assignment operator and of free variables reduces conflicts among cotemporal procedures; the existence of suspensions makes the reduction of form to result more fluid; and the applicative control structure of the model yields a semantics in which the binding of processor to process is natural.

More specifically, all structures consist of three disjoint data types; lists and atoms are manifest results, suspensions are processes in the act of reaching a result. Probing a suspension is equivalent to assigning a priority to a process. But suspensions are also data; evaluation requires the same primitives as data inspection, so the difference between manifest result and process, between process and processor, is blurred.

In the model, a suspension is just a stack, consumed during evaluation to be replaced by a result. Every suspension is referenced uniquely by a list cell; its discovery depends on access to that cell. The processor which uncovers a suspension may ignore it, evaluate it itself, or request that another processor evaluate it. Precisely who evaluates the suspension is immaterial to its eventual value; the result will be the same and no other computation is affected regardless when and how it is coerced. If a second processor is assigned to assist in the evaluation, it needn't dedicate the all its effort to finding the answer; it can evaluate for a while and replace the original suspension with a more advanced one. Processors need never be lost on divergent computations. We impose no hierarchy on processors. A processor can call a second for help, finish its work, and be called by the second processor. We do not preclude the existence of specialized processors for particular tasks, but do require that all processors be capable of fundamental tasks.

Communication

Our goal, then, is the automation of interprocess communication, a significant cost in many operating systems and of the hardware of most CPUs. We have spoken lightly of the "assignment" of processors and of simpifying the scheduling executive, without mentioning a strategy for doing these things. It is not fair to appeal to advances in technology to take care of this problem; it is the critical issue.

Paul Purdom (in conversations) has suggested that a realistic method of interprocess communication is a switching network on the order of the telephone system, permitting bilateral communication between arbitrary processor pairs. This can be implemented linearly in hardware, and provides sufficient generality. Obviously, much study has been put into this kind of system, and numerous papers on multiprocessing alude to it, see the references of Baker and Hewitt [1]. A frame model [17] expresses communication by means of a continuation link, and could use such a system.

A more modest proposal is to have no direct communication structure, and to require that all information be passed through memory. Process requests are sent to a central queue, containing a cell address and a field specifier (see Section 3.4). Processors waiting for something to do fetch these job descriptions directly from the queue, compute for a while and return to the available pool. Whether the effort yielded a value is determined by the calling processor by examination; no messages are sent.

The second proposal has the advantage of being simpler. A communication system on the scale described above requires substantial hardware, although it allows more efficiency. The less ambitious proposal is also easier to expand, as queues are sequential devices. We make no judgement about the mode of communication; it is a design decision. The methods of concurrency described below are independent of the means of passing information.

Opportunities for Concurrency

There are several ways to impose concurrency on our model, but all of them depend on suspensions. Recall that the evaluation of a suspension does not guarantee that a result will take its place, only that the the subsequent suspension, if there is one, is no further from convergence than the first.

1. General Application.

Applicative forms have the syntax:

function_form : argument_form.

If the function_form is a fern, the argument_form is treated as a rectangular array and the elements of the function are applied to its columns. The result is a fern with the same structure as the function form. In the model, construction of the result is supended, but the programmer who uses this construct probably intends to coerce the entire result. Spending some effort to build the manifest value is justified, especially if the function_form is explicit, that is, if second order functionals are not used. In these cases, REST-suspensions are rudimentary; their function is to copy a definitional structure. Time and space can be saved if rudimentary suspensions are bypassed. Furthermore, by allowing FIRST-suspensions to be evaluated, the argument array, which must be sliced into columns remains in active memory due to frequent access.

We predict that using some standard lookahead strategies, such as the pipelining described here, increases speed and space efficiency by reducing environment passing, and ridding structures of rudimentary suspensions. However, the addition of second order functionals to the semantics complicates the issue: REST-suspensions are no longer straightforward, and environment swapping may increase. The same pipelining can be used on the argument side of applicative forms, assuming that different processors are assigned to do function and argument evaluation. Explicit ferns, ferns which are not recursively defined, can be copied completely, suspending elements (FIRSTs) but not lengths (RESTs).

2. Multisets.

The semantics of multisets requires that a means be selected to order them as they are coerced (Section 3.7). Given suspensions, a natural way to do the ordering is evaluate multiset elements concurrently. The fern probe FIRST distributes its resources among the suspended elements until a value shows up. The value is then declared to be the first element of the multiset. The processor in charge may do the evaluation itself or request help from its peers.

Consider this behavior in more detail, assuming that there are three processors, A, B, and C available. Processor A is called to probe a multiset with three suspended elements, and makes four requests to the pool: one for each suspension and one to resume its own task. A itself then returns to the pool. All three processors work on A's four requests, and one of them, say C discovers the probing task. If no value has converged yet, C makes four more pool requests. This continues until at least one multiset element converges, at which time the set is reordered and the probe succeeds.

This approach involves no explicit communication between processors, but there are two forms of implicit iteration. The first is the task request and has already been discussed. The second form results from the fact that multiset transformation cannot take place until all of the supensions are stable. To handle this problem, a validity flag may be required in each cell field, so that competitors can lock each other out. Alternatively, if better communication hardware exists, we may require that each processor notify its caller when it has finished. In the second case, processes are partitioned into three classes: active, suspended, and waiting. The probing processor makes a request for each multiset suspension, then waits to be informed of success. This behavior, called "stinging" [8], is a common feature of operating systems where it is a software construct, and amounts to a process addressable interrupt network [1]. The efficiency of multiset coercion seems to increase with the complexity of the communication system, but again, the semantics of the model is not affected.

3. Lists.

Lookahead, or pipelining, in general application, and concurrency in multisets, are rather straightforward. More exotic forms of concurrency can be adopted for all list structures. We have suggested that that REST-suspensions are usually easy to evaluate. Thus it may be worthwhile to add a "branching heuristic" under which all calls to FIRST lend some portion of their resources to coercing the Rest field of the same cell, and conversely. Also, processors which reduce their suspensions to values without exhausting their resources could spend the remainder of their allotment on neighboring suspensions. The heuristic may even be user specified -- one way to tune the system to particular needs.

The implementation of call-by-need in our model yields both an increase in computational power and a decrease in space when it is used, but often it is not used; by ridding structures of superfluous rudimentary suspensions, data can be localized more readily and expensive context changes eliminated.

4. Work Seeking Processors.

The concurrency discussed so far has been instigated by active processors; little study has been given to the idea that inactive processors can look around for things to do. In the sense that we describe them here, some processors are more important than others. The user's top level evaluator for example, absolutely must come up with something to deliver to the output file. Such processors are recognizable because they have a high priority. Idle processors can be given the duty to seek out highly ranking processors, find nearby suspensions, and evaluate them. This swarming effect may do more harm than good, though; the idle processors are competing for memory access in critical areas, hindering the active one. Hewitt and Baker have addressed this problem [1]. but further study is necessary to establish a viable metric for unsolicited helpers.

5. Special Purpose Devices

It may be assumed that each processor has a local copy of the program it is executing, but these programs need not be identical. There are a number of jobs to which processors may be dedicated, under operator or system control. Input and output require standard transformations of data structures and these processes do not need the full complexity of the evaluator. The storage reclamation algorithm is reference- but not suspension-sensitive. Processes for paging, linearizing, compacting, and encoding data structures increase the efficiency of the system at the critical region of memory access. The system may even include problem specific processors for matrix manipulation, clocking, and so forth.

Summary

We have examined some of the issues of and possibilities for multiple processing in our model. The actual implementation of concurrency depends on the communication protocol between processors, which may be as simple as a single request queue. Most of the opportunities for multiprocessing involve evaluation lookahead, a strategy which trades computational efficiency for decreased startup time. In the spirit of lookahead the extra effort is worth the price because rudimentary suspensions are eliminated quickly. Multiple processors can also be used for breadth-oriented evaluation of multisets, and for dedicated processes. But the number of processors, and the way they are allocated must be independent of the software used for writing algorithms.

Chapter 5. Hardware Considerations

The interpreter is more than a statement of the semantics for a programming language; it is a state machine for hardware implementation. Of course, this can be said of any working program, but in this case the goal of hardware realization has been a major factor in software design decisions. While semantic constructs continue to be added and deleted from the language of the interpreter, its elemental behavior is growing clearer, and attention should be given to the primitive devices this state machine controls. In this chapter, a first attempt is made to specify the fundamental elements of a multiprocessing machine. Some of the goals are well known design problems; others have not been given the attention they deserve. The guiding concept is that the machine must be or appear to be applicative to the user.

There are several justifications for an excursion into the realm of electronics. Hardware design is a product of stepwise refinement; the designer is at liberty to propose any abstract mechanism to solve a problem, but eventually the conceptual machine is adjusted to meet the realities of existing components. We have written a model that works, and further development must in part be guided by physical constraints. We believe our approach to computation to be sound, but a few software examples, complete with exhaustive diagnostic statistics are not very convincing. A concrete demonstration of the principles contained in the interpretor would help attract the attention and interest of industry, as well as the academic community.

It is not unrealistic to propose that an effort be put into building a simple multiprocessor; the expense is not prohibitive if the goals are modest, and may, in fact, be competitive with extensive modeling. Essentially we must show only two things: that the scheduling of individual processors can be independent of software, and that addition of more processors to our system speeds computation.

Memory

Memory access is the one certain critical section of any multiprocessor. To program applicatively and to act on general data structures we require that memory be organized non-sequentially. As is true of the model, we are willing to pay for this organization by dedicating a substantial portion of memory to internal bookkeeping. To the external observer, (i.e. a processor), memory should appear to be a collection of stacks, each of arbitrary length. Each memory unit includes a data distributer or monitor which sends cell contents to processors that request them, manages storage reclamation and arbitrates conflicting requests. Current technology, and possibly the laws of physics, force us to assume that actual memory references are distributed sequentially, at least with respect to a particular memory unit or monitor. The architecture for memory access might consist of queue-like structure; each processor contains its own queue elements, and is supplied by the system with an identification tag. Processors are plugged into the system by means of this queue, and a MEMORY-READ operation might have the following steps.

1. Wait for your queue element to become empty (the system clock causes queue contents to be passed along).

2. Present a memory address and identification tag to queue the request side of your queue element.

3. Look in the data side of your queue element for your identification tag.

4. When your tag appears, fetch the data out of your queue element. (See Figure 5-1).

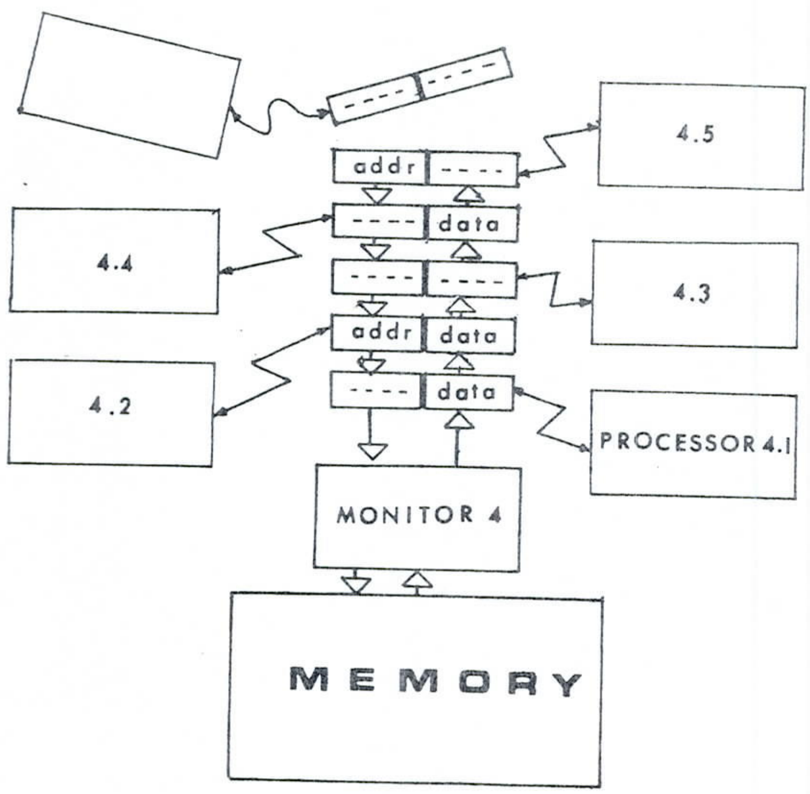


Figure 5-1

A similar architecture could be used for the available processor pool, providing rudimentary communication as discussed in Chapter 5. This kind of mechanism is easily expanded, since processors are connected in parallel. In addition to fetching and distributing data it is the responsibility of the monitor to set semaphonres, return a pointer to the next available free cell, and dispose of discarded data structures.

Evaluators

All monitors, special purpose, and dedicated devices have the same architecture as the evaluator, (Their routines can be written applicatively), except that single-task oriented processes do not require all of the evaluator's power. The basic structure of each processor includes a local copy of the program it is to execute; not a formally defined applicative program, but a microcoded sequential routine. Locality permits instruction fetches to be faster than data manipulation. Microcode is a halfway measure, restricting the system to specific primitives, but allowing some flexibility for development. In addition to the processor executive, there is some local storage, stack caches for example, to reduce the frequency of memory access. Some manifest structures, such as formal parameters and function definitions can be paged into this area as read-only data, so that searches are faster. An appropriate size for both the executive and the local storage area has not been determined.

In its innermost regions, each processor is a register machine with the classic fetch-execute instruction cycle. As a reflection of the model (See Sections 3.5 and 6.5), the following register configuration is sufficient for all evaluation routines:

1. Link.* (one register) When a probe is made on a possibly suspended structure, the instruction address is saved on the recursion stack. When the stack is popped, execution resumes at this point. This is the variable PLACE in the model. As a consequence, all executive routines are at least partially reentrant.

2. Arithmetic operands.**(two registers)

3. Arguments.* (three registers) Eval-procedure arguments are retrieved from the recursion stack into these registers.

4. Values. (two registers) These registers (REVAL and ENVDOT in the model) pass information among eval-procedures. One points to the current environment, the other to the result of evaluation.

5. Inspection. (three registers) These are used to search manifest structures, and do not constitute valid references. Their contents are meaningless between eval-procedure calls.

6. Stacks. (two registers) One for the recursion stack (STACK), the other for the context stack (STAQUE).

Of the thirteen registers mentioned, fewer than nine are actual hardware. Arithmetic operands can be held in inspection registers if an additional bit is included to type-restrict operations. Associated with stack, value and at least one inspection register is a local copy of the cell to which they refer. Registers marked (*) above become subfields of these copies, and are thus logical, not physical entities. Similar to the CDC6600 architecture, loading the pointer part of these expanded registers results in a memory/fetch or store. The instruction set includes conditional branches, determined by cell type or pointer equality loads, register moves, and integer arithmetic. Non-arithmetic operations are byte, or pointer oriented, used to manipulate cell fields.

Assignment to value-registers causes reference count adjustments in memory; all other references are side effect free. Because of the need to account for all references, register-move operations are destructive.

Two constructor routines are implemented in hardware. The first places its argument pointers directly into a new cell; the second uses the first to build suspensions from its arguments, placing them in the new cell. Both constructors effect reference counts.

Algorithms for I/O, storage reclamation, variable association, and so on are either part of or modifications of the evaluation process. In order to keep the local memory requirements small, it is best to separate these routines into individual executives. A portion of central memory is used to hold core images of the microcode programs, to be loaded by individual processors as demands of the system change. Alternatively, these routines can be made reentrant, and shared universally.

We state in Chapter 4 that elaborate interprocess communication is not essential to the semantics of the model; an interrupt network can be replaced by a polling strategy where information is passed through memory. Process requests are made to and extracted from a central pool in the same way that memory requests are made, through some readily expandable queue, or monitor, but in the simpler system, the issue of fault tolerance must be examined. If a processor subcontracts a portion of its work, and if the only way to determine whether the job was done is to examine the contents of a memory cell, then the breakdown of the subordinate processor can procreate throughout the system. Tasks are assigned by passing suspensions which are uniquely referenced. The spectre of deadlock appears when a processor starts manipulating a suspension then malfunctions. The caller is unable to inspect the result and the callee is unable to let it go.

One solution to this problem requires a privileged supervisor to keep account of all processors. The supervisor can override normal memory access protocol, and replace invalidated suspensions with a reserved "malfunction" value, avoiding deadlock. Alternatively, the system can provide for large scale duplication of effort. In this case suspensions cease to be dynamic structures (stacks), and are instead simple function calls: this makes it possible to share them, but opens the way to considerable waste of effort by making it difficult to detach from and resume partial computations.

Summary

The program presented in this paper is a step in the design of a structure oriented general purpose multiprocessing machine. Hardware design begins with the development of a control algorithm, or state machine, which helps the designer establish the architecture for realization. The physical constraints of components cause adjustments in the state machine. This step by step compromise continues until realization is possible.

Our goals for realization are modest; it is sufficient to demonstrate that processors can be added to or taken from the system with no change in software and a minimal change in architecture. With this in mind, we propose a system of independent executives, with communication accomplished by central queues. Memory is organized as a collection of ferns and a significant portion is dedicated to the transparent maintenance of these logical structures. Cell typing is fixed by design and protected from contradictory manipulation. The system provides real-time continuous storage reclamation by including reference counts in all cells.

Processors have a uniform architecture tailored to structure manipulation and simple arithmetic. Each processor may be assigned, at a given time, to any of a number of specialized tasks by loading a particular executive routine. The executives are microcoded programs based on primitives which manipulate generalized binary graphs. Optimization of executives is directed toward the minimization of memory access.

Chapter 6. Implementation Notes

Section 6.1 Introduction

The interpreter/model is implemented in PASCAL on the Indiana University CDC6600. In this section the topics discussed in Chapter 3 are reexamined, with more emphasis on the implementation code and the developmental aspects of the model. The program is not a finished product; as a vehicle for computation it is cumbersome at points and lacks some of the standard conveniences of established interpreters; and as a semantic model it is incomplete at points (most notably, the interface with the file system is not suspended). Opportunities for optimization abound, but in anticipation of further development, clarity takes precedence over efficiency.

Chapter 3, an overall look at program behavior, Chapter 6, an examination of the current implementation, and Chapter 7, suggestions for improvement, are together a top level document of the program. The code itself is found in Appendix C. We adopt the following spelling conventions for keywords:

1. Specific interpreter function names and PASCAL variable names are upper case: FIRST, DEFINE, . . .
2. Cell types begin with upper case: Atom, Multiset, . . .
3. Cell fields are lower case, underlined: cdr, pname, . . .
4. PASCAL procedures and keywords are upper case and underlined: EVAL, MEMORY, . . .

Section 6.2 Cells

MEMORY is an array of records called NODEs (or cells) which are of uniform size and designed to fit into a sixty bit word (the CDC6600 word size). The records have two formats, depending on whether the cell is to be used as a print name (type Pname) or not. Figure 6.2-1 is the PASCAL declaration.

```
TYPE    PTR = 0..MAXPTR;
        NUM = -MAXNUM..MAXNUM;
        REFERENCE = PACKED RECORD
                CASE NUMBERP : BOOLEAN OF
                 FALSE:(RR:PTR);
                 TRUE:(NN:NUM)
                END; (* REFERENCE *)
        NODE = PACKED RECORD (* FITS IN A 60-BIT WORD *)
                MULTI:BOOLEAN;    (* MARKS LIST AS ORDER BY CONVERGENCE *)
                ATOMP:BOOLEAN;    (* REDUNDANT-CHECK CDR.PNAME *)
                EXTRA:0..7;       (* NOT USED-SAVE FOR WAITE-SHORR *)
                CASE PNAME : BOOLEAN OF
                     FALSE:(REF:REFERENCE;
                            CAR:REFERENCE;
                            CDR:REFERENCE);
                      TRUE:(LENGTH:0..7;
                            EXTRAA:0..7;
                               CHR:PACKED ARRAY[1..8] OF CHAR);
                END; (* NODE *)
```

Figure 6.2-1

The content of a Pname includes the codes for the characters of a Literal Atom. A three bit field, <u>length</u>, is set aside to specify the length of the Atom (Figure 6.2-2).

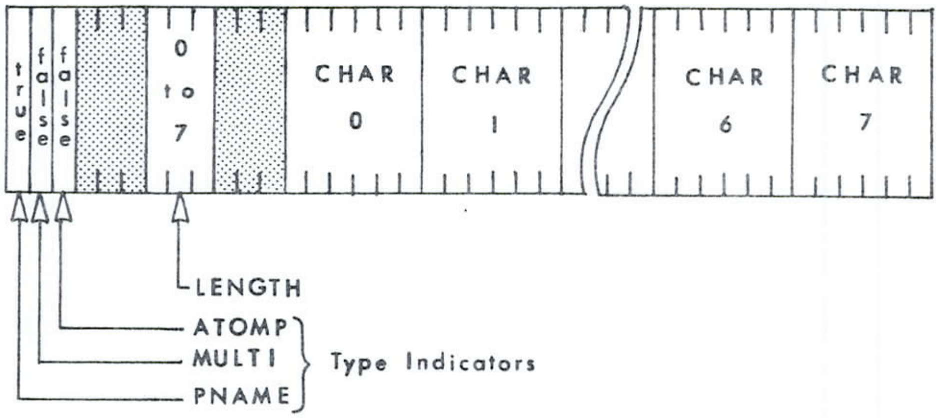


Figure 6.2-2

Cells which are not Pnames contain three pointer fields called ref, car, and cdr. A flag is associated with each field to specify whether it is to be used as a pointer or a number. A pointer field, together with its specification bit, form a subtype called a REFERENCE. (Figure 6.2-3).

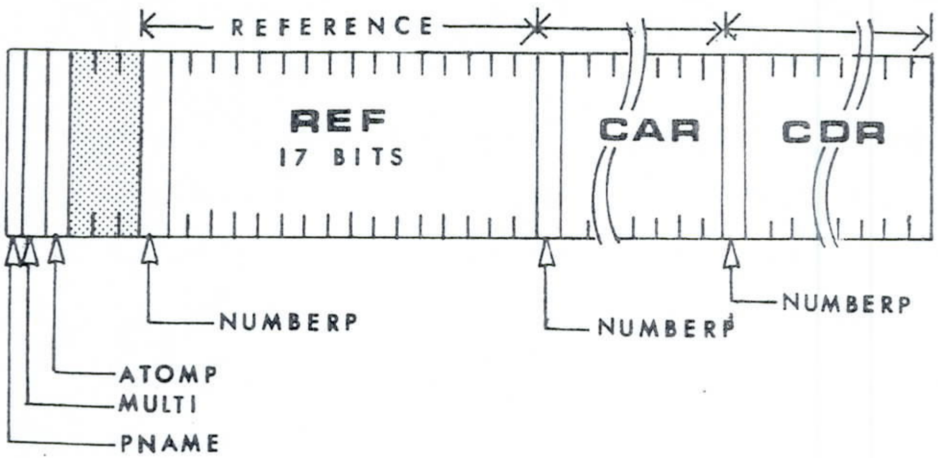


Figure 6.2-3

All cells have three additional marking bits for type specification, but their interpretation is context dependent, as summarized in Table 6.2-5, and when the context permits, the flags atomp, multi, and pname can be used for data. As in Chapter 3, cellular representation of data structures follow the conventions: a) Pnames are not shown, and Atomic values appear in boxes; b) Fern cells are divided into thirds, showing ref, car, and cdr from left to right; and c) Suspensions are divided horizontally and include the suspended form in lower case. Convergent field values are sometimes moved into the Fern cell field which points to them.

Figure 6.2-4 shows the cell representation of the fern (3 (BLIND mice)). Cells C1 and C2 comprise the top level List structure, whose first element is the Atomic cell C5. The value of the Atom C5 is 3; this value could have appeared in the car field of cell C1. Cell C3 heads a two element Multiset (denoted by the mark in the upper left corner). The second element is a suspension whose form is "mice". If it is coerced, this suspension will converge to MICE. The reference counts of Fern cells is given, showing in this example that there are external references to the interior of the List.

The PASCAL restrictions on subtype access are not protective. One may ask for the cdr of a Pname, even though this cell type has no cdr field. A sequence of three character codes is returned. One is free to interpret REFERENCE field contents as either pointer or integer, regardless of the value in numberp, and there are occasions when it is tempting to do so. The system programmer will avoid these temptations as a matter of principle. Pointers are not numbers and should not be treated as though they were; they cannot be added, or negated; the only acceptable operations on pointers are assignment and comparison for equality. Neither are numbers pointers. If it becomes necessary to examine a MEMORY location according to the integer contents of a cell field, the numberp flag should be changed explicitly to FALSE for the sake of clarity.

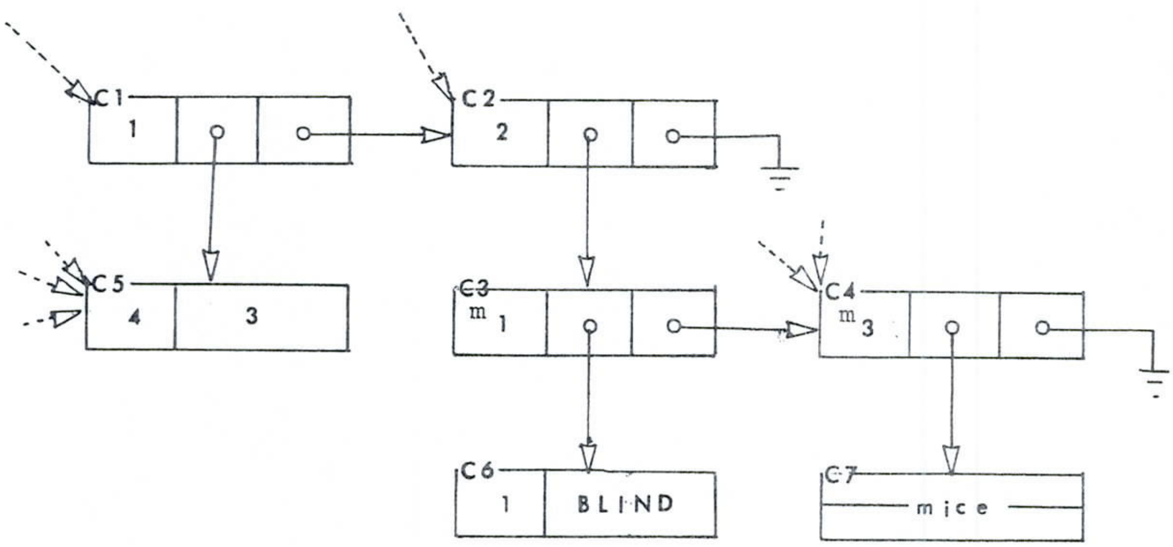


Figure 6.2-4

Field assignments are done with the PASCAL operator, ':=', and can take place on the cell or REFERENCE level. The program initialization provides templates for cell types (e.g. NEWSUSPEND, NEWATOM) in which the type specifiers are established. Thus, the fields of an active cell can be replaced individually, or changed as a whole. For example:

```
VAR   ENODE:NODE; (*working storage*)
      X,Y,Z:PTR;   (*pointer variables*)
BEGIN
      ENODE:=NEWSUSPEND; (*set type flags*)
      ENODE.CAR.RR:=X;
      ENODE.REF.RR:=Y;
      MEMORY[Z]:=ENODE  (*cell assignment*)
      MEMORY[Z].CDR.RR:=Z  (*field assignment*)
END
```

TABLE 6.2-5
Cell Type Indicators

	Flags	Comments
	P M A R C C N U T E A D A L O F R R M T M . . . E I P # # #	"#" means numberp. "x" indicates flag value is irrelevant.
1.	T x x x x x	A Print Name when the cell is pointed to by an Atom. In the PRINT algorithm, pname = TRUE indicates that cdr contains the traversal thread. In an evaluation CONTEXT, pname = TRUE indicates that the current suspension came from a car field.
2.	x T F T F F	Multiset: ref = reference count. car = FIRST element. cdr = REST fern.
3.	x F F T F F	List: ref = reference count. car = FIRST element. cdr = REST fern.
4.	F F T T T x	Numeric Atom: ref = reference count. car = Numeric value. cdr - not used.
5.	F F T T F F	Literal Atom: ref = reference count. car = next Atom in hash bucket. cdr = points to Print name.
6.	F F F F T F	Suspension: ref; usually an argument pointer. car; usually an encoded eval-procedure name. cdr; usually the stack link.
7.	F F F F F F	Suspension: FALSE in ref.numberp specifies type Suspension. car can also be a pointer.

Section 6.3 Memory Management

This section looks at the Memory management kernal, the procedures NUDGE, NEWNODE, DISPOZE, and RECYCLE. These routines lend modularity to the program, maintain the integrity of cell types, and protect the legitimacy of the free space stack, AVAIL. DISPOZE and NEWNODE are essentially AVAIL operators for pushing and popping. RECYCLE does an extended push operation for returning large structures. AVAIL is linked through cdrs.

Initialization takes place in the main body of the program (Figure 6.3-1). All value-registers are global and are set to NIL at the outset to insure that there are no superfluous references at point (a) in the code. Next, the OBLIST is established, then the remainder of MEMORY is linked sequentially to establish AVAIL, (c).

```
a(* INITIALIZE REGISTERS *)
EXP:= NIL; ENVDOT:= NIL; ASSOCVAR:= NIL; ASSOCLIST:= NIL;REVAL:= NIL;
DMY:= NIL; DMY1:= NIL; FULFILL:= NIL; PRS:= NIL; VARB:= NIL;
VAL:= NIL; FN:= NIL; ARGS:= NIL; LST:= NIL;P:=NIL; Q:= NIL;
FP:= NIL; AF:= NIL; CARFN:= NIL; CDRFN:= NIL; AEXP:= NIL;STACK:=NIL;
(* INTITIALIZE MEMORY *)
      AVAIL:=OBLISTSIZE+1;
      MEMORYLIMIT:= MEMORYSIZE;
     bFOR I:=1 TO OBLISTSIZE DO MEMORY[I]:= NEWCONS;
     cFOR I:= OBLISTSIZE+1 TO MEMORYSIZE-1 DO
        BEGIN
        MEMORY[I]:= NEWCONS;
        MEMORY[I].CDR.RR:= I+1
        END;
      MEMORY[MEMORYSIZE]:= NEWCONS;
      MEMORY[0]:= NEWCONS;    (* ESTABLISH NIL *)
```

Figure 6.3-1

The procedure NUDGE, (Figure 6.3-2) increments the reference count of its argument. Cells without reference counts are protected from the side effect. Also excluded is NIL, which is treated as a special case by the management routines.

```
PROCEDURE NUDGE(P:PTR);     (* UPS THE REFERENCE COUNT OF FERN CELLS *)
      BEGIN
      IF (NOT (P=NIL)) AND MEMORY[P].REF.NUMBERP THEN
         MEMORY[P].REF.NN:= MEMORY[P].REF.NN+1
      END;
```

Figure 6.3-2

DISPOZE (Figure 6.3-3) first insures that its argument is neither NIL nor a Literal Atom, then clears the contents of the appropriate cell and pushes it onto AVAIL. The reference count is not examined; DISPOZE is called only when the cell's count is known, for example, when the cell is a stack element (see Section 6.4). As suggested in the commentary, further optimization of the program may produce "well-structured" system objects, such as generalized Print names, which can be traversed and added to AVAIL quickly. DISPOZE is the place to recover such specialized structures.

The procedure RECYCLE (Figure 6.3-4) is structure sensitive, and unlike DISPOZE, considers reference counts. It ignores NIL, at point (a) in the Figure, then obtains a working copy of the cell to be recycled. If the reference count of this cell exceeds one, it is decremented and nothing more is done (b). Otherwise, if the cell is Atomic it is DISPOZEd, at (c). If (a), (b), and (c) fail, then the cell is either a Suspension or an unreferenced Fern, and in either case its cdr is a pointer. The cdrs are followed and the same tests, (a'), (b'), and (c'), are repeated on the next cell in the structure. The process continues until a test succeeds, and the resulting cdr list is attached to AVAIL. The appended structure has not been fully recycled; substructures in the car and ref fields have not been checked, but this is taken care of in NEWNODE.

```
PROCEDURE DISPOZE(N:PTR);    (* RETURN A NODE TO AVAIL*)
     VAR VISIT:NODE;
     BEGIN
     IF (N=NIL) THEN BEGIN END
     ELSE IF MEMORY[N].ATOMP AND (NOT MEMORY[N].CAR.NUMBERP) THEN
          BEGIN
          (* INTERN RECOVERS LITERALS *)
          DRETURNS:= DRETURNS+1;
          MEMORY[N].REF.NN:= 0
          END
     (* THERE MAY BE OTHER 'SUB-ATOMIC' STRUCTURES, E.G.
        SOME STACKS MAY BE RECOVERABLE AS A WHOLE.  THIS
        IS TAKEN CARE OF AT THIS POINT. *)
     ELSE
          BEGIN
          VISIT:= NEWCONS;
          VISIT.CDR.RR:= AVAIL;
          MEMORY[N]:= VISIT;
          AVAIL:= N;
          DRETURNS:= DRETURNS+1
          END
    END;
```

Figure 6.3-3

The function NEWNODE returns the top element of AVAIL (Figure 6.3-5). If AVAIL is NIL, free space is exhausted, and NEWNODE HALTs the program. Before returning the cell to the calling routine, its car and ref fields are RECYCLEd, and then reinitialized. This protects the MEMORY structure, and the caller is free to use the initial NIL pointers.

```
PROCEDURE RECYCLE ( P:PTR );        (* FOLLOWS CDRS UNTIL A HIGH
                                     REFERENCE COUNT.  THE RESULT
                                     GOES TO AVAIL.  NEWNODE
                                     RECYCLES THE OTHER FIELDS *)
    LABEL 1,2;
    VAR CURSOR,TRAILER:PTR; CNODE:NODE;
    BEGIN
  a IF (P=NIL) THEN GOTO 2;
    IF (TRACE>10) THEN WRITE("RECYCLE ",P:1,":");
    CURSOR:= P; CNODE:= MEMORY[P];
  b IF CNODE.REF.NUMBERP AND (CNODE.REF.NN>1) THEN
         BEGIN
         IF (TRACE>10) THEN WRITENODE(CURSOR,TRUE);
         CNODE.REF.NN:= CNODE.REF.NN-1;
         MEMORY[P]:= CNODE
         END
  c ELSE IF CNODE.ATOMP THEN DISPOZE(P)
    ELSE
         BEGIN
         WHILE TRUE DO
              BEGIN
              RETURNS:= RETURNS+1;
              IF TRACE>10 THEN WRITENODE(CURSOR,TRUE);
              TRAILER:= CURSOR;
              CURSOR:= CNODE.CDR.RR; CNODE:= MEMORY[CURSOR];
           a' IF (CURSOR=NIL) OR (CURSOR=TRAILER) THEN GOTO 1;
           b' IF CNODE.REF.NUMBERP AND (CNODE.REF.NN>1) THEN
                   BEGIN
                   CNODE.REF.NN:= CNODE.REF.NN-1;
                   MEMORY[CURSOR]:= CNODE;
                   GOTO 1
                   END;
            c' IF CNODE.ATOMP THEN
                   BEGIN
                   IF (TRACE>10) THEN
                        BEGIN
                        WRITENODE(CURSOR,FALSE);
                        WRITELN("<ATOM>")
                        END;
                   DISPOZE(CURSOR);
                   GOTO 1
                   END
              END;   (* WHILE LOOP *)
 1:      MEMORY[TRAILER].CDR.RR:= AVAIL;
         IF TRACE>10 THEN WRITELN("--> AVAIL");
         AVAIL:= P
         END;
 2:      END;
```

Figure 6.3-4

```
FUNCTION NEWNODE:PTR;        (* RETURN THE FIRST NODE ON THE AVAIL
                                LIST.  IF AVAIL IS EMPTY WE'RE
                                OUT OF MEMORY.  RECYCLE THE GARBAGE
                                IN THE CAR AND REF FIELDS *)
    VAR RESULT:PTR; VISIT:NODE;
    BEGIN
    IF NOT (AVAIL=NIL) THEN
        BEGIN
        RESULT:= AVAIL; VISIT:= MEMORY[RESULT];
        AVAIL:= VISIT.CDR.RR;
        IF NOT VISIT.REF.NUMBERP THEN RECYCLE(VISIT.REF.RR);
        IF NOT VISIT.CAR.NUMBERP THEN RECYCLE(VISIT.CAR.RR)
        END
    ELSE
        BEGIN
        WRITELN; WRITELN;
        WRITELN(" -=>-=>-=> MEMORY IS EXHAUSTED. ");
        WRITELN("      -=> YOU HAVE SPECIFIED ",MEMORYLIMIT:1," NODES, ");
        WRITE("AND THE LIMIT IS ",MEMORYSIZE:1,".");
        WRITELN;WRITELN;WRITELN;HALT
        END;
    MEMORY[RESULT]:= NEWCONS;
    NEWNODE:= RESULT
    END;
```

Figure 6.3-5

Literal Atoms, and their associated Pnames are not reclaimed by the kernal routines. Literals are chained into hash buckets and removing them requires a search to update the chain links. However, the Function INTERN, which is responsible for making all OBLIST entries, is reference count sensitive, and Atoms with zero counts are replaced by new Literals when they are found.

One additional procedure is described here because it is mentioned in the next Section, and while it is not specifically a MEMORY management procedure, it represents a system primitive of equal priority. SETREG performs value-register assignment (See Section 3.2), along with the necessary reference count adjustments. Its argument, REG, is called by reference, and a test is included to avoid extraneous MEMORY access (Figure 6.3-6).

```
PROCEDURE SETREG(VAR REG:PTR;VAL:PTR);   (* THIS IS A SYSTEM ASSIGNMENT
                                            STATEMENT WHICH INCLUDES
                                            THE ASSOCIATED ADJUSTMENTS
                                            IN REFERENCE COUNTS. *)
    BEGIN
    IF (TRACE>5) THEN WRITELN("               ASSIGNMENT: ",VAL:1,".");
    IF NOT (REG=VAL) THEN
        BEGIN
        NUDGE(VAL);
        RECYCLE(REG);
        REG:= VAL;
        END
    END;
```

Figure 6.3-6

Section 6.4 Stacks and Assignment

One of the four pairs of stack operations to be found in the program is discussed in this section, to demonstrate that the abstract type STACK is not compromised. We use the pair PUSHONE and POP which manipulate suspensions. These procedures are local to the procedure EVAL, and operate on the variable STACK, which is treated as a global. STACK is always assigned to the active suspension in the current context. STACK's entries may contain either two pointers or a pointer and a number; PUSHONE and POP are used in the latter case.

This yields a programming invarient for calls to these operations:

> POP is called only if the top entry in STACK contains a number and a pointer item, or equivalently, only if the most recent addition to STACK was done by PUSHONE.

This invarient can be established by inspection of the procedure EVAL. Suspensions are implemented by linking type Suspension cells through their cdr fields. Both operators use a type NODE variable, ENODE, for working space. With few exceptions, the number-item is an encoded eval-procedure name, and the sequence:

```
. . .
PUSHONE(n,a);
POP;
. . .
```

is the method used to call the eval-procedure n with argument a.

Figure 6.4-1 is the code for PUSHONE. ENODE is used to construct the STACK entry; the integer item is placed in car, the argument pointer in ref. Since ref contains a pointer, ENODE is type suspension. A new

MEMORY cell is obtained, linked to STACK and the contents of ENODE are copied into it. Containment in a suspension entry constitutes a reference, so the argument is NUDGEd.

```
PROCEDURE PUSHONE(NUM: INTEGER; PNT: PTR);
                (* THERE ARE TWO STACK-PUSH ROUTINES.  THE
                   FIRST PUSHES A PROCEDURE CALL (CAR) AND
                   A POINTER (REF).  THE STACK IS LINKED BY
                   THE CDR FIELD. *)
        (* GLOVAR STACK (REFERENCE); GLOCON NEWSUSPEND (A TEMPLATE)
           GLOBAL PROCEDURE NEWNODE,NUDGE *)
        BEGIN
        ENODE:= NEWSUSPEND;
        ENODE.CAR.NUMBERP:= TRUE;
        ENODE.CAR.NN:= NUM;
        ENODE.REF.RR:= PNT;
        NUDGE(PNT);
        ENODE.CDR.RR:= STACK;
        STACK:= NEWNODE;
        MEMORY[STACK]:= ENODE;
        IF (TRACE>5) THEN
            BEGIN
             WRITE("      PUSH-1: ",STACK:1,"--> ");
            WRITENODE(STACK,TRUE)
            END;
        END;   (* END OF PROCEDURE PUSHONE *)
```

Figure 6.4-1

POP is the inverse operation on STACK (Figure 6.4-2). ENODE is used to get a working copy of STACK's top; this reduces the MEMORY accesses to one. To avoid parameter binding, the globals EXP and PLACE are introduced to receive the field contents of ENODE. PLACE is type INTEGER and is therefore an inspection-register. EXP is a value-register, so the code must account for its change in reference. Instead of calling the value-assignment operator SETREG to change EXP, its contents are RECYCLEd explicitly, and EXP is given its new value directly with a PASCAL assignment.

In this way, the argument reference which was recorded during PUSHONE is "stolen" for EXP, and a MEMORY access is avoided. Since STACK is next DISPOZEd, its contents are erased and there is no inconsistency.

```
PROCEDURE POP;           (* POPS A PUSH OF TYPE ONE *)
     (* GLOVAR PLACE: INTERGER; EXP, STACK: PTR *)
     BEGIN
     ENODE:= MEMORY[STACK];
     PLACE:= ENODE.CAR.NN;
     RECYCLE(EXP);           (* REUSE THE REFERENCE FROM THE STACK *)
     EXP:= ENODE.REF.RR;
     DISPOZE(STACK);
     STACK:= ENODE.CDR.RR;
     IF (TRACE>5) THEN
          BEGIN
          WRITELN(" POP:");
          WRITE("           PLACE: ",PLACE:1,", EXP: ",EXP:1);
          WRITE(", ENV: ",ENVDOT:1,", REVAL: ",REVAL:1);
          WRITELN(", MODE: ",MODECODE:1,", STACK: ",STACK:1)
          END
     END;   (* END OF PROCEDURE POP *)
```

Figure 6.4-2

The PUSHONE-POP pair satisfy the normal prerequisites for the abstract type stack: they agree in the link field and in the item specifications. However, POP is not a failsafe operation; it does not verify that STACK is non-empty (not NIL) before it fetches the top element. When the context does not insure STACK's integrity, it is the programmer's responsibility to verify the legality of POP. This is true of all operator pairs.

Section 6.5 Evaluation

The procedure EVAL is large, and its behavior is complex when traced, but it is highly modular. The macroscopic behavior of EVAL is described in Section 3.4; here we look more closely at the mechanics of eval-procedure calls. Let us assume that a context has been established, that is, that STACK has been assigned to a Suspension, and that ENVDOT points to the Suspension's environment. Further, suppose that the top STACK element is a call to the eval-procedure TOP, which is the analog of the LISP function EVAL. We begin our observation at the top of the innermost evaluation loop (LOOP in Section 3.4). Figure 6.5-1 shows the code at the beginning of this loop. At (a) in the figure, the STACK POP takes place. The inspection-register PLACE receives the encoded eval-procedure name, in this case, TOP (see Section 6.4). The value-register EXP is assigned to TOP's single argument; suppose that it is the Literal Atom FOO. Following the POP, a PASCAL CASE statement is entered; the CASE labels are the eval-procedure names, and branching is determined by the contents of PLACE. As a result, execution begins at point (b). If the global TRACE is sufficiently large (Section 6.8) a message is printed.

EVALuation begins. EXP is compared with NIL at (c), and the test fails. Next the contents of the MEMORY cell to which EXP refers are examined. It is determined that EXP is not a Numeric (d), but that it is a Literal Atom (e). It is found by inspection that EXP is not a special system Atom, which brings us to the point (f), where an environment search is instigated. The environment is saved on the STACK (ASSOC is destructive); REVAL is set to initialize the search; a recovery routine is pushed in case the search fails, then ASSOC is called.

```
aPOP;
 WHILE NOT ALLDONE DO
 BEGIN
 CASE PLACE OF
    bTOP:
         BEGIN
         IF (TRACE>3) THEN WRITELN("TOP   ");
        cIF EXP=NIL THEN
              SETREG(REVAL,NIL)
        dELSE IF MEMORY[EXP].CAR.NUMBERP THEN
              SETREG(REVAL,EXP)
        eELSE IF MEMORY[EXP].ATOMP THEN  (*THE EXPRESSION IS AN ATOM*)
              BEGIN
              IF (EXP=QHSH) THEN SETREG(REVAL,EXP)
              ELSE IF (EXP=JAWS) THEN EVALERROR(0,NIL)
              ELSE IF EXP=QSPEAK THEN SETREG(REVAL,MAKENUM(SPEAK))
              ELSE IF EXP=QSTOP THEN
                   BEGIN
                   ALLDONE:= TRUE;
                   FINIS:= TRUE
                   END
              ELSE IF EXP=QENV THEN SETREG(REVAL,ENVDOT)
             ELSE IF EXP=QUNDEFINED THEN SETREG(REVAL,JAWS)
              (* INSERT OTHER SYSTEM ATOMS HERE *)
             fELSE
                   BEGIN    (* SEARCH THE ENVIRONMENT FOR A BINDING *)
                   PUSHONE(RESTORE,ENVDOT);   (* ASSOC DISECTS ENVDOT *)
                   SETREG(REVAL,QUNBOUND);    (* INITIALIZE ASSOC *)
                   PUSHONE(LOOK,EXP);         (* IN CASE OF ERROR *)
                   PUSHONE(ASSOC,EXP)
                   END
              END
         ELSE
              BEGIN   (* THE EXPRESSION IS A LIST.   *)
              IF EXP=MEMORY[EXP].CDR.RR THEN
                   BEGIN              (* THE EXPRESSION IS STARRED *)
                   PUSHONE(STARRED,EXP);
                   PUSHONE(CAR,EXP)
                   END

                      ORY[EXP].CAR.RR;  (* GRAB THE FUNCTION *)
                       Y[EXP].CDR.RR;
                         THEN

                             ) THEN

                                 XP);
                                 MORY[PT1].CAR.RR].CAR.NN
```

Figure 6.5-1

ASSOC is a collection of four eval-procedures. The first and part of the second are shown in Figure 6.5-2. The details of association are discussed later; for now, we will suppose that the formal variable FOO is found immediately. ASSOC has broken the environment into pieces which it sends as arguments to ASSOC1, at (a). At (b), ASSOC1 discovers FOO, and is to return the car of the actual parameter list, but since this field may be suspended, the eval-procedure CAR is called, at (c).

```
ASSOC:      (*  THE ENVIRONMENT IS A LIST OF
                    FORMAL PARAMETER-ARGUMENT PAIRS.
                    SEARCH EACH PAIR FOR A BINDING.
                    EXP IS THE VARIABLE NAME.
                    THE CALLING ROUTINE MUST SAVE THE
                    ENVIRONMENT. *)
    BEGIN
    IF (TRACE>3) THEN WRITELN("ASSOC ");
    IF (REVAL=QUNBOUND) THEN
        BEGIN
        IF (ENVDOT=NIL) THEN
            BEGIN
            SETREG(EXP,SEARCH(ALIST,EXP));
            IF MEMORY[EXP].MULTI THEN SETREG(REVAL,CADR(EXP))
            END
        ELSE
            BEGIN
            PUSHONE(ASSOC,EXP);
          a PUSHTWO(EXP,CAAR(ENVDOT));
            PUSHONE(ASSOC1,MEMORY[MEMORY[ENVDOT].CAR.RR].CDR.RR);
            SETREG(ENVDOT,MEMORY[ENVDOT].CDR.RR)
            END
        END
    END;


ASSOC1:
    BEGIN
    IF (TRACE>3) THEN WRITELN("ASSOC1: ");
    LOAD(ASSOCVAR,FP);     (* EXP IS THE ACTUAL PARAMETER LIST *)
    IF (FP = ASSOCVAR) THEN SETREG(REVAL,EXP)
    ELSE IF ISATOM(FP) THEN SETREG(REVAL,QUNBOUND)
    ELSE IF ISATOM(EXP) THEN EVALERROR(7,ASSOCVAR)
    ELSE
        BEGIN
        PT:= MEMORY[FP].CAR.RR;
       bIF (PT = ASSOCVAR) THEN cPUSHONE(CAR,EXP)
        ELSE IF ISATO
            BEGIN
            PUSHT
```

Figure 6.5-2

CAR is shown in Figure 6.5-3. We are concerned with the case in which the desired result is suspended, at (a) in the figure. Suspension sensitivity is confined to the Fern probes. Upon finding a suspension, CAR calls itself, then does a context push on STAQUE. The current suspension is set aside, replaced by the new one. If that computation succeeds. its result will be returned as a binding for FOO.

```
CAR:        (*THIS IS THE USER CAR.  EXP HAS THE NODE
                 WHOSE CAR IS TO BE RETURNED.  CHECK
                 FOR A SUSPENSION.  RETURN THE CAR
                 IF IT ISNT SUSPENDED, ELSE EVALUATE IT. *)
    BEGIN
    IF (TRACE>3) THEN WRITELN('CAR ');
    IF ISATOM(EXP) THEN EVALERROR(9,EXP)
    (** ELSE IF RESERVED(EXP) THEN
        BEGIN
        PUSHONE(CAR,EXP);
        MODECODE:= MODECODE-1
        END **)
    ELSE
        BEGIN
        PT:= MEMORY[EXP].CAR.RR;
        IF (PT=JAWS) THEN
            BEGIN
            IF MEMORY[EXP].MULTI THEN KICKLIS(EXP)
            ELSE EVALERROR(0,NIL)
            END
        ELSE IF NOT SUSPENDED(PT) THEN
            BEGIN
            SETREG(REVAL,PT);
            (** CANCEL(EXP) **)
            END
      a ELSE IF NOT MEMORY[EXP].MULTI THEN
            BEGIN
            PUSHONE(CAR,EXP);
            CONTEXTPUSH(EXP,TRUE,ALLOCATE(1));
            (** CANCEL(EXP) **)
            END
        ELSE
            BEGIN
            IF (TRACE>3) THEN WRITELN('-=>-=>-=> MULTI <=-<=-<=-');
            KICKLIS(EXP)
            END
        END
    END;
```

Figure 6.5-3

This brief and superficial example of evaluation is included to give a flavor of EVAL's behavior. Three features are noteworthy:

1. Evaluation is accomplished through a series of eval-procedure calls, made by pushing the procedure name on STACK, (PUSHONE).

2. Arguments are passed either directly with the value-registers REVAL and ENVDOT, or by STACK references. In the later case, the value-register EXP is assigned automatically by POP, which does the procedure calls. Additional arguments are pushed in pairs by PUSHTWO and retrieved into value-registers by LOAD.

3. Only the eval-procedures CAR and CDR are Suspension sensitive. Other eval-procedures do field access only when the data structure is known to be manifest; otherwise the probes are called in case coercion is necessary.

We now describe, in a bit more detail, two sets of eval-procedures, EVLIS and ASSOC. EVLIS is a straightforward linear recursion, ASSOC recurs on both fields of its arguments. The reader is invited to follow the descriptions in the code (Appendix C).

All user List structures are implicit calls to EVLIS, whose formal definition is:

```
DEFINE EVLIS (LIST ENVIRONMENT)
   if NULL:LIST then NIL
  else CONS:〈EVAL:〈FIRST:LIST ENVIRONMENT〉
            EVLIS:〈REST:LIST ENVIRONMENT〉〉.
```

Since CONS is suspended, a call to EVLIS results in the creation of a single cell. If EVLIS was user defined the resulting cell would have been dominated by a new environment binding LIST to EVLIS's argument as well as saving the evaluation environment. To avoid the extraneous environment EVLIS is broken into two eval-procedures:

1. EVLIS: The list is passed via the STACK push; the environment resides in ENVDOT. If the list is empty, NIL is returned to the calling routine via REVAL. Otherwise, EVLIS is called with one argument, the list, and CDR is called to coerce the list's cdr field.

2. EVLIS1: The environment is still in ENVDOT. REVAL now points to the coerced cdr of the list. REVAL is assigned to
   CONS:⟨FCAR:list EVLIS:REVAL⟩
   under ENVDOT. FCAR is a system function (see Section 3.4), used for suspension sharing, equivalent to EVAL: 1:list

ASSOC does not lend itself so readily to formalism, but as with EVLIS, the possibility of suspensions causes ASSOC to be broken into a set of procedures. Because it is not linear, there are four pieces:

1. ASSOC: The environment is a list of pairs. Each pair consists of two lists, a formal parameter structure and an argument. ASSOC passes the pairs successively to ASSOC1, seeking a value for its argument. As a last resort the ALIST, a totally manifest structure, is searched.

2. ASSOC1: ASSOC disects the formal-actual pair into two arguments, EXP and FP, and passes them to ASSOC1. If a binding is found, it is returned in REVAL. If the formal structure is exhausted, the calling routine is informed of failure. Otherwise, there are two possibilities: if the car of the formal structure is atomic, the cdr of the actual argument is coerced and ASSOC2 is called; if the car is a structure, ASSOC2 is called to search the two cars, and ASSOC3 to search the two cdrs.

3. ASSOC2 takes two arguments, a variable and a formal structure. REVAL contains the coerced actual structure. These three entities are passed to ASSOC1. This is the recursive call.

4. ASSOC3 coerces the cdr of the actual argument structure, then calls ASSOC1.

In summary, the evaluator is driven by two stacks, STACK and STAQUE. STACK is a current active suspension controlling the order of eval-procedure calls, and containing procedure names and arguments. Eval-procedures are broken into pieces according to the need to probe suspended structures. The probes are empowered to replace the current STACK with a new suspension by manipulating the STAQUE of contexts. Values are carried between contexts and eval-procedures in the value-registers REVAL and ENVDOT.

There are three pairs of stack operations local to EVAL:

1. PUSHONE/POP -- for manipulating STACK. Items: a number, an encoded eval-procedure name, and a pointer, the eval-procedure's first argument. POP chooses the proper eval-procedure and assigns EXP to the argument item.

2. PUSHTWO/LOAD -- for manipulating STACK. Items: two pointers, the second and third arguments to eval-procedures. LOAD's arguments are value-registers, called by reference. These registers are assigned to the argument items.

3. CONTEXTPUSH/CONTEXTPOP -- for manipulating STAQUE. Items: a Fern cell, and a flag, and the current STACK. The flag specifies which suspended field is to take control. Resources are allocated for the subsequent computation. Leftover resources are stored in the vacated cell field. CONTEXTPOP occurs when resources are exhausted, or when a value has converged. The cell field is assigned to the value, and the old STACK regains control.

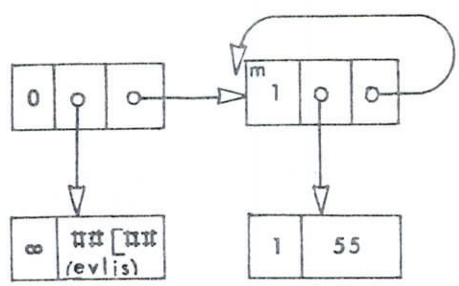
Section 6.6 Input

The procedure MYREAD converts a character string into cellular data structures. Characters are fed to MYREAD by function GETCH, which fetches them from the input file. All non-atomic forms are list structures; the user brackets are macro characters for calls to the function LIST; the colon is parsed into a call to APPLY. For example:

1. ⟨A B C⟩ becomes (##⟨## A B C)
2. ADD1:5 becomes (##:## ADD1 5)
3. DEFINE TEST X X. becomes (DEFINE TEST X X)

The hashmark is used in macro names because the user cannot construct atoms with that character. Star formations are better expressed graphically:

4. [55 *] becomes



MYREAD is driven by a stack called RSTACK, whose associated operations are RPUSH and RPOP, and which is linked through the ref field. Whenever possible, the RESULT of MYREAD is constructed from discarded RSTACK cells. The architecture of this procedure is similar to that of EVAL (See Section 6.5); a collection of RSTACK manipulators is gathered into a CASE statement. These read-procedures are selected by the input characters. There are also a number of service routines; these are discussed first.

The procedure RESTART initializes the parser and RECYCLEs RSTACK. It is called in the event of a syntax error, or when the user cancels the form he is building with a double slash, "//". When a syntax error is discovered RERROR is called. It prints a message and calls RESTART.

RLOOK does single character lookahead, primarily to determine if the infix operator APPLY is next in the input. RCOMP is also involved with the APPLY operator, and compresses successive calls to it into the proper list form.

RBUILD is a pseudo constructor, responsible for the RSTACK-to-form conversion. It is called when the current character is a closing bracket. This bracket terminates a structure residing at the top of RSTACK. The structure is popped and sent to RBUILD. The new RSTACK top is a partial structure to which RBUILD enqueues the RESULT. Using RSTACK to build results directly necessitates the ref linkage.

Each call to RPUSH starts the construction of new substructure. Each call to RBUILD completes a substructure and resumes construction of the one above it. The read-procedures, whose calling order is determined by the input stream are summarized here:

If the input character is a . . .

1. alphabetic character:
   a. Construct a Literal Atom; assign RESULT to it.
   b. If RESULT is "NIL" assign RESULT to NIL.
   c. If RESULT is "DEFINE", RPUSH(DEFINE).
   d. RLOOK for a colon. If one is found, RPUSH(APPLY).
   e. RBUILD.

2. digit:
   a. Construct a Numeric Atom.
   b. RLOOK for a colon; if one is found, RPUSH(APPLY).
   c. RBUILD.

3. " " or "[" or "(":
   a. RPUSH(## ##) or RPUSH(##[##) or RPUSH(##(##)

4. " " or "]" or ")":
   a. If unbalanced then RERROR.
   b. Assign RESULT to RPOP, then RBUILD.

Section 6.7 Output

The PRINT routine is contained in the procedure READLOOP, and consists of a next of WHILE-TRUE-DO loops, whose hierarchy is shown in Figure 6.7-1. Unmentioned in the Figure is PRINT's sensitivity to starred structures which is straightforward. The algorithm is essentially that of Friedman and Wise, presented in Output Driven I/O [Friedman & Wise, 1976,7],translated into standard PASCAL. As mentioned in Section 3.6, PRINT maintains value-registers in order to automate storage reclamation; these registers are called P, Q, and STACK for compatibility with the Friedman-Wise version.

A recursion stack is avoided by back-threading the argument structure. The strategy is the same as is used in Schorr-Waite garbage collection. [11, Section 2.3.5, Algorithm E], except that no marking takes place. The thread determining bit (ATOM in [11]) is placed in the pname field.

The variable name STACK is somewhat misleading, since it refers to a structure different than the abstract type stack, described in Section 3.2. STACK's sole purpose is the reconstruction of the argument structure, which takes place if there are external references. Entries therefore, have reference counts; they are threaded and not linked uniformly through a particular field.

In this section we concentrate on the interface between PRINT and EVAL, a feature not discussed by Friedman and Wise. The problem here is one of resources and the development of the model with respect to multiprocessing. On the one hand, the evaluator expects to be given a finite limitation on its computation; on the other, PRINT must force coercion in order to come up with output, and its resources are essentially infinite. But PRINT

READ LOOP.

1. Fetch data from MYREAD.

2. If the result is EXIT, HALT READ LOOP.

3. If the result is atomic, print it and start READ LOOP.

4. PRINT LOOP. Print "(".

    1. CAR LOOP.

        1. EVALuate the car.

        2. If the result is atomic, print it and start CDR LOOP.

        3. Thread the car field.

    2. CDR LOOP.

        1. EVALuate the cdr.

        2. If the result is NIL, print ")" and start POP LOOP.

        3. If the result is atomic, print it and start POP LOOP.

        4. If the reference count is more than one, thread the cdr and start POP LOOP.

        5. Otherwise, restart PRINT LOOP.

        6. POP LOOP.

            1. If STACK is NIL, restart READ LOOP.

            2. Follow cdrs and reconstruct the list.

            3. At the first car thread restart PRINT LOOP.

Figure 6.7-1
The PRINT algorithm.

should not be so special as to preclude other I/O routines from the model. Still, the model is in an early stage with respect to multiprocessing and the code for PRINT-EVAL communication gives little more than passing consideration to the question of suspensions:

```
. . .
. . .
SETREG(P,Q);
SETREG(Q,NIL);
SETREG(REVAL,PROCESS(CAR,P));
repeat
       Q:=REVAL;
       REVAL:=NIL;
       EVAL(Q,37)
until NOT SUSPENDED (REVAL)
Q:=REVAL;
REVAL:=NIL;
. . .
. . .
```

The first two calls to SETREG move the contents of the value register from Q to P and clear Q for interaction. PROCESS is called to create a suspension whose value is FIRST:P: REVAL is assigned to the suspension. EVAL is called repeatedly on this suspension until it converges, with a finite resource limit. The direct assignments inside the loop are essential: since REVAL points to a Suspension, a cell with no reference count, SETREG would RECYCLE it regardless of the value of Q. Those assignments after the loop have similar justification. The resource allocation is arbitrary; any positive integer will do.

In the program PRINT has the last word on computation. But there is no reason why this algorithm, which is just a character string builder, cannot be treated like other constructors and suspended. One can imagine, in the extreme, a coin slot installed on the output display, providing resources for PRINT to pass on to EVAL.

Section 6.8 Multisets

Five routines are involved in the evaluation of multisets: the eval-procedures CAR, CDR, KICKAR, and KICKDR, and the service routine KICKLIS, which does the structure transformation described in Section 3.7. CAR and CDR are cell-type sensitive and call KICKLIS when given a Multiset argument. KICKAR and KICKDR are auxiliary probes, a means to enforce uniform evaluation on Multiset structures; they are called by KICKLIS.

Briefly, when CAR encounters a Multiset it checks the first cell for a convergent element. If a suspension is discovered instead, the Fern is passed to KICKLIS, which searches for a convergent element deeper in the structure. If no convergent element is found, KICKAR is called once for each cell. KICKAR does nothing if the car of its argument has converged. Otherwise a fixed amount of resources is consumed evaluating the suspension there. When all of the cells have been KICKed, CAR makes another pass.

The transformation algorithm has three phases. In the first, a search is made for a convergent element. If one is found, the second phase is executed during which the element is moved to the front of the FERN. If all elements are suspended, the third phase is executed, and calls to KICKAR are made. During the phase one search, KICKLIS reverses the structure as it goes. The other phases restore the original order. The disadvantages of this attack are that the code is a bit convoluted, and that parallel transformations are restricted more than they need be. The advantages are that lookahead is avoided during restructuring, and that the Multiset cells are KICKed in the same order as they have in the structure.

Assume that CAR has been called on the Fern shown in Figure 6.8-1.

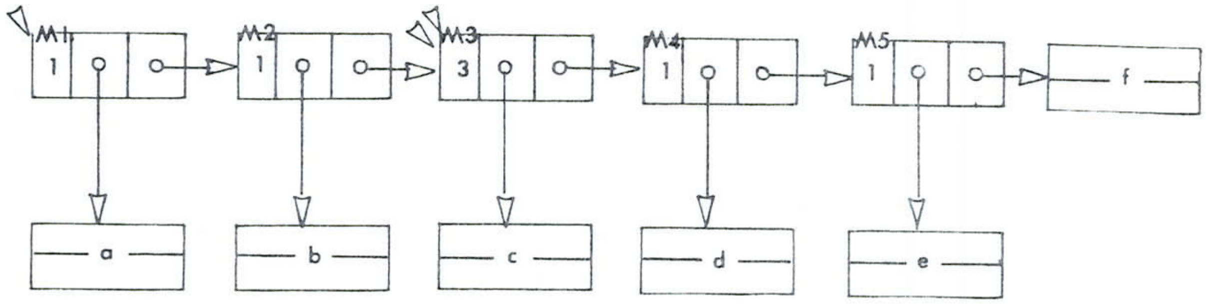


Figure 6.8-1

KICKLIS traverses the Fern, reversing as it goes, until the suspension f is found: (Figure 6.8-2)

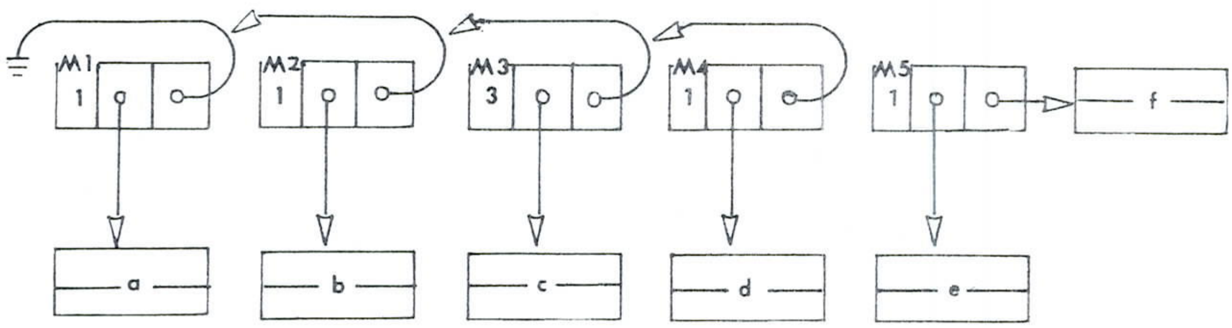


Figure 6.8-2

Since no convergent element was uncovered, a call is made to KICKAR for each cell in the structure, as well as one to KICKDR for f: (Figure 6.8-3)

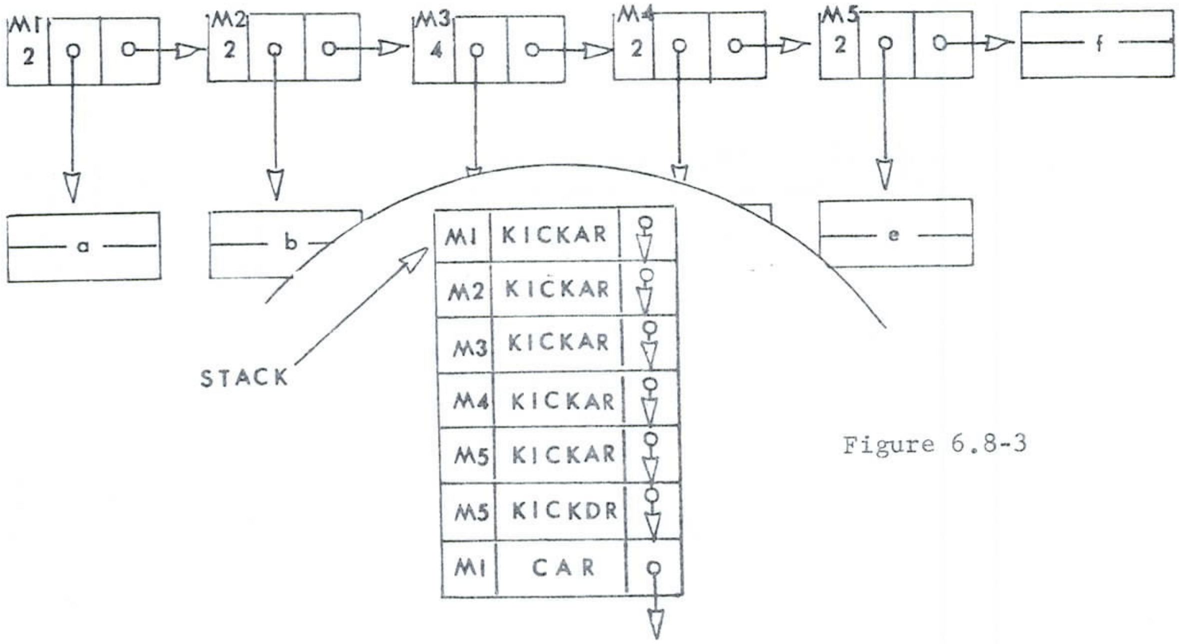


Figure 6.8-3

The stack references have incremented each of the Fern cells' reference counts. The KICKs are executed, and the process is repeated until a suspension, say d, converges. CAR again calls KICKLIS, and the Multiset is searched up to cell C4.

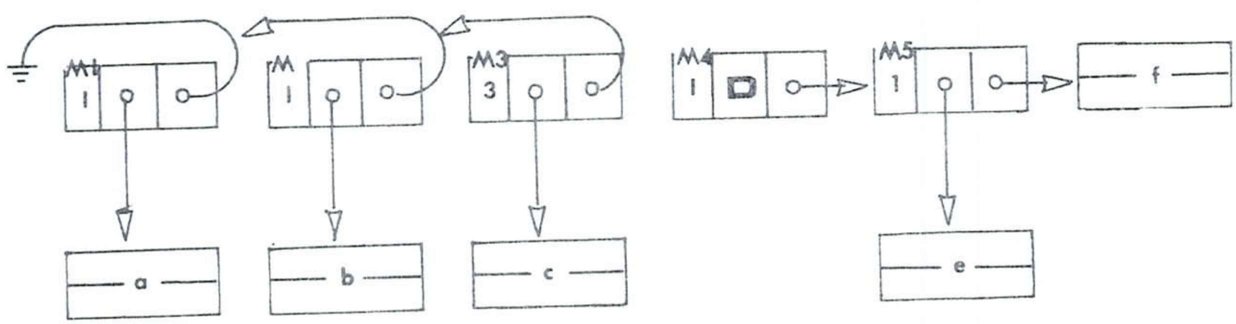


Figure 6.8-4

The Multiset is restructured; D is moved to the front of the Fern. (Figure 6.8-5)

KICKLIS's search continues until a suspension, a convergent <u>car</u>, or a List cell is found (See Section 3.7). It is clear that two copies of this version of the algorithm cannot be permitted to transform the same Fern at the same time. This is not currently a problem in the model, since KICKLIS is not suspendable. If multiple processing is allowed, however, some provision must be made for simultaneous access to multisets.

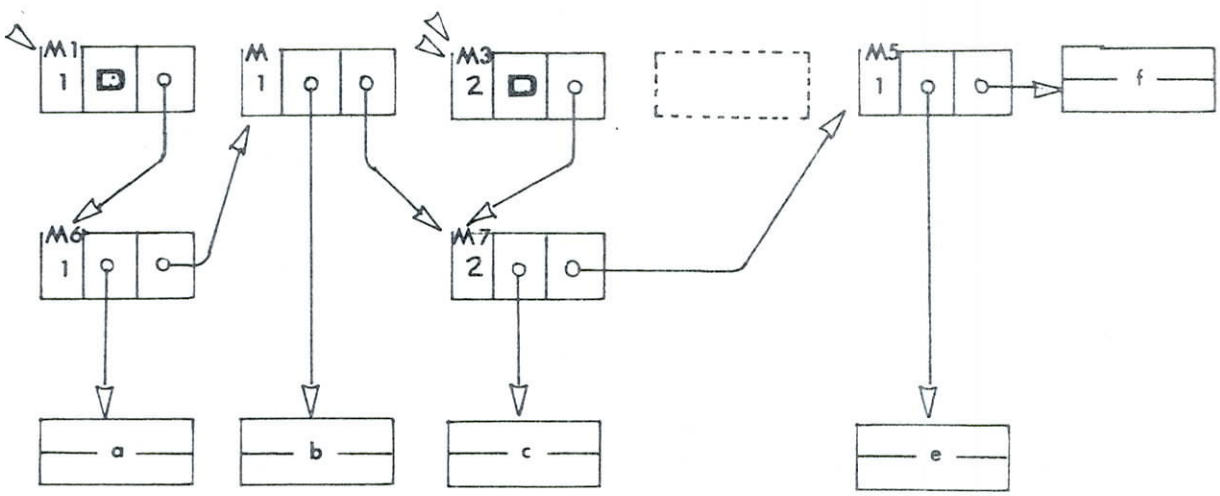


Figure 6.8-5

Chapter 7. Suggestions for Continued Development

Section 7.1 Optimization

Issues of code improvement are complicated by the program's dichotomy of purpose. On one level the interpreter is a computational tool, an expression of the applicative philosophy, to be _used_ for problem solving. From this point of view, the program is both too large and too slow; a good deal of work is needed to polish the code and to weed out excessively costly features. As a programming language the interpreter needs user oriented facilities like those found in LISP: debugging aids, external file access, and so on. But the program is also a software model, a contribution to machine design; in the author's opinion, this is its primary purpose. In this sense, optimization means further development along the lines of multiprocessing. Minimal cost algorithms for interpreter primitives do not lead necessarily to optimal hardware design.

The following sections are a potpourri of topics dealing with the code as it is, and as it might be improved. The issues, and the projects they imply range from trivial adjustments to abstruse generalities; the simple goals serve as a practical introduction to the code for anyone wishing to further this project; the more general discussions are academic in intent, attempts to resolve conflicts of opinion about proper computational behavior.

Section 7.2 Cell Representation

Table 6.2-5 shows that there are seven cell types differentiated by six flags. In fact there are more types than seven, but the type indicators can still be encoded into fewer bits. At the software level, the type decoding consumes time, but in most type driven algorithms, several flags are checked. By moving all the flags into an integer field and declaring constant descriptors, alternative constructs are replaced by faster and more readable CASE statements. For example, the sequence:

```
WORKINGSPACE:= MEMORY[X];
if WORKINGSPACE.MULTI then ALTERNATIVE1
else if WORKINGSPACE.CAR.NUMBERP then ALTERNATIVE2
else if NOT WORKINGSPACE.REF.NUMBERP then ALTERNATIVE3;
```

could be replaced by

```
  case MEMORY[X].CELLTYPE of
  MULTISET:   ALTERNATIVE1;
   NUMERIC:   ALTERNATIVE2;
SUSPENSION:   ALTERNATIVE3
  end;
```

Encoding the type indicators raises no conflicts with the hardware model, it would be done there too, but there is some danger of programmer carelessness. The NUMBERP flags are a constant reminder that References are a subtype and that pointers are not numbers. Furthermore, in the multiprocessing model, it may become necessary to manipulate some of these bits individually to lock out conflicting cell access.

It is unlikely that the PASCAL field extraction primitives yield the fastest access to cell fields. While the cells are designed to fit into a single CDC6600 word, the compiler reserves about two words for each cell. A relatively easy and extremely important project is to hand pack the cells and to write customized access functions.

Associating a one-bit semaphore, or busy bit, with each cell has been suggested as a way to arbitrate simultaneous access in a multiprocessing environment. All processors should be allowed to read the contents of a cell at any time, but parallel writes must be prohibited. Multiset transformation (Section 6.8) is a good example; if the traversal process marks each cell it visits, others are informed that they cannot do the transformation themselves, but with read privileges they can watch the progress of the evaluation. A Boolean function RESERVED(pointer) might be added to the model, which returns TRUE if the semaphore of its argument cell is on. Otherwise, FALSE is returned and the bit is turned on. The inverse operation CANCEL(pointer) releases the cell to other probes. The state of being reserved constitutes another cell type, and the system programmer must adopt the RESERVED/CANCEL protocol in the code of all routines which side affect active cells.

Projects having to do with cell representation

1. Encode the cell type indicators into an integer field. This is an easy task but considerable changes in the code are necessary.

2. Hand pack the cells and write machine dependent access functions. Changes in the code can be made with an editor macro.

3. Examine the issue of simultaneous writes. Expand the model to simulate multiprocessing by time-slicing independent evaluators. Develop statistics concerning memory access. This is a thesis level project.

4. Propose and model alternatives to the semaphore scheme.

5. Write a memory dump routine, and examine the relationships between suspended and manifest structures, the displacement between a cell and its successors.

Section 7.3 Memory Management

Every call to NEWNODE is followed by explicit field assignments to the new cell. Until the programmer is confident of the program behavior, these statements are helpful, but it would be an improvement if NEWNODE did the assignments itself. Arguments to NEWNODE are: a type specification for the cell, three references, and a code stating whether the references should be NUDGEd.

PASCAL record fields cannot be called by reference, so field assignments are made using the assignment operator, ':=', followed by direct calls to RECYCLE and NUDGE. While the user is not permitted to make field changes in active cells, the system programmer does it all the time. Procedures called RPLACA, RPLACD, and RPLACR, analogous to SETREG, should be added to the management kernal since they are likely hardware primitives.

Since storage reclamation is concurrent with computation, the median evaluation time is slower than it would be if a garbage collector was used. Machine level implementation of RECYCLE and DISPOZE should be given high priority. In particular, RECYCLE uses few enough local variables to fit in the the CDC6600 register file, and its speed could be greatly increased if it were made machine dependent. There is more about RECYCLE in Section 7.4.

Linking stacks yields a universality in structure manipulation, but evaluation traces show that a lot of time is wasted by popping a stack, DISPOZEing the top node, immediately fetching the same cell, and pushing it back on the stack. Each instance of the evaluator should have a local array of cells to absorb this inefficiency. Stack cells could be aquired and returned in batches.

Projects having to do with memory management

1. Rewrite NEWNODE to do field assignments in the new cell. This is a fairly easy project.

2. Implement register assignment, RPLAC- operations, NUDGE, DISPOZE and NEWNODE at machine level.

3. Add an evaluation STACK cache to the model. This project requires sound knowlege of evaluation behavior.

4. Add a reference cache to NUDGE, which avoids two memory accesses when a Fern cell or Atom is NUDGEd and then immediately RECYCLEd.

5. Rewrite INTERN and DISPOZE to permit arbitrarily long print names for Literal Atoms. By doing this the cell type and special format (Section 6.2) for print names is eliminated; the substructure of atoms becomes a a normal suspension, interpreted as a list of character codes.

6. Change the semantics of Numeric Atoms to allow arbitrarily large integers or real numbers.

Section 7.4 Storage Reclamation

In RECYCLE, the argument is searched linearly (through the cdr field) and the resulting list is appended to the AVAIL stack. Since the list may be of any length, the threat of a garbage collector-like pause in computation is not eliminated, assuming that a single processor is involved. An optimal RECYCLE must run in constant time even if doing so involves extra memory accesses. Constant time is realized with a short search; in practice only a few cells contain three pointers. All Fern cells and most stack cells have a number in at least one field; the object of the search is to move part of the RECYCLEd structure from the cdr field to one of these numeric fields.

If the argument to RECYCLE is a Fern, its cdr is moved immediately to its ref, and the Fern is pushed onto AVAIL. NEWNODE takes care of the substructure. Otherwise, the argument is a Suspension, a stack element having two or three pointers. In this implementation, it can be proven by inspection that stack entries with three pointers never occur in sequences; by following at most one link, a cdr can be moved to some other field. Program traces demonstrate that RECYCLE rarely visits more than four cells. For this reason, the best algorithm would search a bit longer than necessary, in the hope that moving a field will not be necessary.

Section 7.5 Evaluation

Substantive changes in EVAL alter the semantics of the language; they are made only after careful consideration of the consequences. Since the model was first implemented, EVAL has undergone two complete revisions and numerous adjustments, but there is a great deal more to do. This section gives some representative examples.

There is a considerable need for more sophisticated error recovery. At one extreme there are still instances of programmer error that cause program abortion. They must all be eliminated, but finding them is a matter of chance. More to the point, a cohesive strategy must be found to provide to the user sufficient information about recoverable mistakes. Suspensions make this a difficult problem; in the case of multisets, evaluation errors do not terminate computation at all, they cause more concentration on other suspensions. EVAL is, in a sense, desensitized to programmer mistakes, but when a point is reached where further progress depends on illegal results, evaluation stops and the programmer should be informed,to as full an extent as possible, what went wrong and where it happened. The atom #BOTTOM# represents a noticably divergent computation, and the eval-procedure TOP makes a test for it just as it does for NIL. One approach to improving error diagnostics is to associate with every occurence of this atom a list of function names, or similar messages, which accumulates until it is returned at top level.

EVAL dedicates a lot of time and space to functional combination. When a fern is found in the function position of an applicative form, it is scanned twice to check for star configurations and null entries. If an infinite list shows up the argument is coerced far enough to determine if the appropriate vector is starred too; if it is, the result is starred. The cost of checking for such unusual conditions makes one wonder if they should be included in the semantics, or if the construct should be weakened to save resources. One solution is to eliminate functional combination from the core of the evaluator and to superimpose a system function for general application, sensitive to ferns. This function is analogous to the LISP COND which has been absorbed into our language. A reasonable compromise is to restrict the star configuration to be a lexically defined entity, that is, to permit the user to build star structures only at top level from the keyboard. If this is done, infinite functions are constrained always to be manifest, and their search need not be coercive. Whenever suspension-sensitivity is avoided, the result is simpler and less expensive eval-procedures.

The kind of change described above restricts the eventual transition to second order in the interpreter. In fact, some limited generalization of function application is permitted now: atoms bound to numbers can be applied, and a special FUNARG form is included in the implementation. It seems a short step to full generality; provisions for full EVALation of the function part of a form can be added without much effort (this is also true of free variables). In general, opinions about such facilities are not well formed. There is something suspect about unlimited generality in applicative programming, it clouds structured thinking, and the changeover should be postponed until it becomes obvious that it is necessary.

Projects having to do with the evaluator.

1. Change TRACE so that error messages are supressed.

2. Provide the user with a MESSAGE facility. MESSAGE takes two arguments; the first is not evaluated and is printed directly on the display in

the midst of evaluation, and the second argument is evaluated and returned as MESSAGE's value. MESSAGE permits the user place checkpoints in his/her program.

3. Provide the user with a BREAK facility, which permits limited examination of the environment at any point during evaluation.

4. Allow the user, by setting a switch, to have free variables.

5. Expand the second order behavior of EVAL.

6. Devise a more informative error recovery scheme.

7. Change the semantics of function definition to allow nested conditional expressions.

Section 7.6 I/O

To date, not much attention has been paid the input and output routines of the program, and a great deal of work needs to be done in this area. Both from the user point of view, and that of the program as a model, numerous changes have already been proposed, and on the whole the problem of I/O requires further study. In the most general terms, the interpreter is just a program which reduces one character string to another; where the first string comes from, and where the second goes, has been left up to the PASCAL compiler and the KRONOS operating system. The fact that there will be more than one input source, that a particular instance of the evaluator may require more than one input string, and that output may be sent to several destinations, imposes a need for generalization of the I/O routines.

From the user's aspect, there is an urgent need for file manipulation primitives. It should be possible, during interactive sessions, to retrieve function definitions from external files, and to save functional environments for repeated use. A simplistic way to provide such capabilities is to declare a number of special I/O files in the program heading. Much more elegant is a set of system request procedures by which fern structures are transformed into permanent file requests. GETCH and its output counterpart PUTCH are altered to place their character strings onto variable files.

From the modelling aspect the problem goes deeper. Neither MYREAD or the PRINT loop use suspended constructors. It is essential to the semantics of the interpreter that this be changed, particularly as regards input. The character "string" is instead a "stream" [Landin, 1966], an infinite sequence of elemental data. The input constructor, RBUILD, should be strict in its first argument, conceptually. To make this change requires substantial alteration of the eval-procedure TOP, where it is currently assumed that the top level form is manifest in both fields. TOP's examination of input forms must be made coercive.

Similarly, changes are needed in PRINT, which constructs a character stream from a sequence of ferns and atoms. PRINT has no explicit pseudo-constructor, like RBUILD; it produces its result with PASCAL output statements, and so is not suspendable in its present form. Furthermore, PRINT, like KICKLIS, restructures its argument as it goes. Because of this it needs the same kind of lockout provisions as the multiset evaluator: two instances of PRINT can't have access to the same fern.

There is a uniform, if expensive way to put the problems of MYREAD and PRINT on the same level. The goal is to separate both functions from any dependence on PASCAL primitives,, to treat the I/O streams as ferns by turning them into ferns. To do this, an intermediary is imposed, via I/O management routines, between the file system and the interpreter. The character string is converted directly to a list structure whose elements are character-atoms. This structure is cdr suspended in the top level environment, which is altered to include an association function from logical device names to physical system files. This additional superstructure is expensive in both space and time, but is a reasonable approach to I/O in the model.

It can be expected that I/O devices of the future won't be the passive machines that they are today; they will assume the responsibility to transform raw input into generalized data structures as proposed here, and that they will have direct access to memory as independent processors. As we state in the beginning of this section, this aspect of computation needs a lot of study, but for the present, attention should be paid, in the model, to the problem. The best way to do this is to implement virtual models of particular devices, in order to study their behavior.

Projects having to do with input and output

1. Create a special function like TRACE which takes one atomic argument. Extract the print name from this atom and load it into a type alpha array. The resulting file name is sent to an externally compiled COMPASS subroutine, say GETFILE, which makes a system request to the operating system to retrieve the named file from the user's permanent file space. The resulting file is declared to be the source of input for the interpreter, which reads forms from it and evaluates them. When the file is exhausted, input reverts to the keyboard.

2. Create two special functions, as in project 1, which dump and retrieve the top level environment to local or permanent files. In this way repeated definition of debugged functions is eliminated between sessions. Consider encoding strategies by which these core images can be compressed into a minimum of space.

3. Place I/O intermediaries between the interpreter and the PASCAL I/O routines which convert character strings to list structures. Include in the representation a way to associate character streams with various virtual devices, so that the model can simulate the presence of several keyboards, for example.

4. Make MYREAD suspended.

5. Provide diagnostic routines to dump function definitions in some legible form. Write a simple definition editor so that the user can alter function meanings.

6. Expand the multiprocessing model so that I/O executives use the same coercive protocol as the evaluator.

Section 7.7 Multisets

The evaluation of multisets is the most recent addition to the program, and the implementation can be criticized at several levels. The primary objection so far is that the multiset probe traverses the entire manifest structure before any suspensions are evaluated. This is an un-recursive approach; the structure may be long, tying the probe up when it could be contributing to the convergence of an element. My contention is that since the multiset is manifest, the search isn't prohibitively expensive. Moreover, since it is generally agreed that multiset probes must lock out their peers when they do the transformation, it is better to confine this phase to a single period. Once the probe knows what suspensions are involved in evaluation, it can let go of the Fern; other probes can search the structure while suspension evaluation is taking place.

In the code of KICKLIS, the double reversal of the Fern is confusing, but comprehensible. There is, however, a more subtle reason not to do the reversals. If a probe examines a multiset and discovers that it can't do the restructuring itself, it can at least add its resources to the element evaluations. In order to do this, the probe follows the links of the fern and evaluates suspensions as it goes (if a convergent value is found it can be returned!). If the multiset is reversed, this cannot be done.

Section 7.8 Miscellaneous Projects

1. Provide more system arithmetic functions like ADD1 and PLUS.

2. Make AND and OR system functions.

3. Implement tail recursion by associating with each context the name of the function being applied. In some recursive forms, lookahead can allow the environment to be altered instead of replaced.

4. Give the user carriage control ability, so that he can format output.

5. Add a system function which makes it possible to load and execute library routines.

6. Model a memory paging scheme.

7. Add two new cell subtypes, StrictF and StrictR, which force coercion in one of the fields. Strict functions are discussed in [Friedman and Wise, 1977].

8. Modify the program to model continuation passing. Under this scheme, contexts contain a continuation link through which they inform their callers of successful coercions.

9. Determine the feasibility of adding a heuristic field to multiset cells, a rule by which the multiset probe distributes resources among suspended elements.

References

1. H. G. Baker, Jr. and C. Hewitt. The incremental grabage collection of processes. Proc. ACM Symp. on Artificial Intelligence and Programming Languages. Sigplan Notices 12, 8; SIGART Newsletter 64 (August, 1977), 55-59.

2. E. W. Dijkstra. A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ (1976).

3. D. P. Friedman. The Little LISPer. Science Research Associates, Palo Alto, California (1974).

4. D. P. Friedman and D. S. Wise. An environment for multiple-valued recursive procedures. In B. Robinet (ed.), Programmation, Dunod Informatique, Paris (1977), 182-200.

5. D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner (eds.), Automata, Languages and Programming, Edinburgh Univ. Press, Edinburgh (1976), 257-284.

6. D. P. Friedman, D. S. Wise, and M. Wand. Recursive programming through table look-up. Proc. ACM Symp. on Symbolic and Algebraic Computation (1976), 85-89.

7. D. P. Friedman and D. S. Wise. Output driven interpretation of recursive programs, or writing creates and destroys data structures. Information Processing Lett. 5, 6 (December, 1976), 155-160.

8. D. P. Friedman and D. S. Wise. The impact of applicative programming on multiprocessing. IEEE Trans.on Computers (to appear). Also Proc. 1976 Intl. Conf. on Parallel Processing (IEEE Cat. No. 76CH1127-OC), 263-272.

9. D. P. Friedman and D. S. Wise. Aspects of applicative programming for file systems. Proc. ACM Conf. on Language Design for Reliable Software, SIGPLAN Notices 12, 3 (March, 1977), 41-55.

10. D. P. Friedman and D. S. Wise. Nondeterminism with referentail transparency. Technical Report No. 64, Computer Science Department, Indiana University, Bloomington, (1977).

11. D. E. Knuth. Fundamental Algorithms (2nd. ed. 2nd printing), Addison-Wesley, Reading MA (1975).

12. P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. Comm. ACM. 8, 2 (February, 1965), 89-101.

13. P. J. Landin. The next 700 programming languages. Comm. ACM. 9, 3 (March, 1966), 157-162.

References (cont.)

14. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, MA (1962).

15. J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg (eds.), Computer Programming and Formal Systems, North-Holland, Amsterdam (1963), 33-70.

16. M. Wand. Continuation-based program transformation strategies. Technical Report No. 61, Computer Science Department, Indiana University, Bloomington (1977).

17. M. Wand and D. P. Friedman. Compiling lambda expressions using continuations. Technical Report No. 55, Computer Science Department, Indiana University, Bloomington, (1976).

APPENDIX A

A User Manual for The Interpreter

Section A.1 Introduction

This is a basic manual for a programming language called the Interpreter. The fundamental features of the Interpreter's syntax, the rules of the language, are presented here for the introductory student, or for anyone unfamiliar with the applicative approach to programming. The interpreter is interactive; users can write short programs at a terminal to be executed immediately. The reader should study this manual once, then go over it again, trying the examples out on the computer. The Interpreter is still very much in the experimantal stage of development. Your comments about its behavior, about this document, and about applicative programming in general are gratefully received. When problems arise, please be patient. Because it is constantly being modified, there are often errors in the Interpreter's code. Feel free to ask for assistance; if a program fails to work it is not necessarily the fault of the programmer.

Section A.2 Data

There are two kinds of data and that is all. <u>Atoms</u> are numbers or character strings, like 5 and 762 and SIX and WATER. As their name indicates, Atoms should be thought of as indivisible nuggets of information. They cannot be broken down into smaller pieces; they are not composed of smaller "building blocks". There is no special relationship between the Atoms COLD and GOLD, for example, because they rhyme or because they have the same second letter. The Atom 5 cannot be said to be closer or more similar to the Atom FIVE than is the Atom 6, even though 5 and FIVE represent the same human concept.

Such relationships can be expressed, however, by "tying" Atoms together into <u>Lists</u>, the other kind of data. Lists are collections of data put in order. <1 2 3 4> is a List, whose elements are the Atoms 1 first, then 2, then 3, then 4. One often hears that Sets are collections too, but a List is not a Set, because Sets are not ordered. There are other differences between Sets and Lists. The same element can appear twice in a list,

<2 SWEET 2 BE 4 GOTTEN>

is a List with six elements, two of which are the Atom 2. Moreover, Sets are said to be equal if they have the same elements, and this is not necessarily true of Lists:

<1 2 3> is not the same as <1 3 2>.

<5> *may* not be the same as <5>.

There are some similarities, though, between Lists and Sets. Just as the elements of a Set may themselves be Sets, the elements of a List may be Lists.

<<TWO 2> <SEVEN 7>>

is a List with two elements, each of which has two elements. In this List we can say that the Atoms 2 and TWO are "related" because they are elements of the same element, but such relationships are in the mind of the beholder and are not automatic.

The shortest known List is <>, which has no elements. Only one such List exists; we sometimes refer to this unique empty List by its name, *NIL*. Lists are members of a more general collection of structures called *Ferns*. We have more to say about Ferns later, but briefly they are collections of data without specific order. Most of what we say about Lists applies to Ferns as well.

The memory of the Interpreter consists of cells, and when the occasion calls for it, data structures are pictured in terms of the cells which form them. Figure A.2-1 is the cell representation for the List <<2 TWO> <7 SEVEN>>.

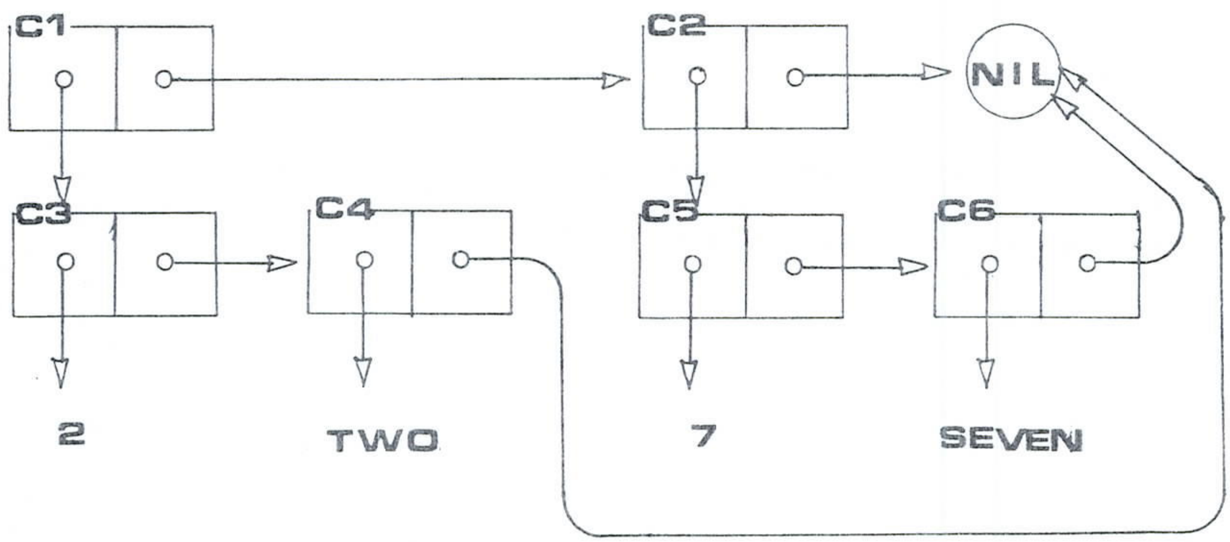


Figure A2-1.
Cell representation

Cells C1, C2, C3, C4, C5, and C6 are List cells. Every List cell contains two arrows; one (the left one) points to the first element of the List the cell represents, the other to the List that is left when the first element is removed. The first element of the List C1 is the List C3. C3's first element is the Atom 2. The List C3, without its first element is the List C4, whose first element is TWO. If TWO is removed, the List that is left is <>.

Atoms are character strings. Numbers are strings consisting only of digits; Atoms which are not numbers may consists of digits and letters, but they must begin with a letter:

SAM, R2D2, and C3PO are _Literal_ Atoms.

-64000,... 0, 1,...,64000 are _Numeric_ Atoms.

REVIEW OF SECTION A.2

1. The Interpreter is based on DATA.
    a. Atomic data consists of Literal Atoms, character strings starting with a letter, and Numeric Atoms which are integers. (See footnote)
    b. Lists are elements of a larger class of data structures called Ferns. They provide a way to gather data into collections.

2. Lists are collections of data, but they are not Sets.
    a. Lists have order.
    b. Two elements of a List may be the same.
    c. Lists with the same elements are not necessarily the same.
    d. The unique empty List (Fern) is called NIL.

3. There are two ways to express Lists.
    a. The bracket notation: <1 2 <3 4>>.
    b. The cell notation, which is used to discuss the Interpreter's internal representation of data.

NOTE: In the current implementation of the Interpreter, Literal Atoms are restricted to have fewer than nine characters.

Section A.3 Evaluation

The interpreter is a program which transforms data. At the top level it receives data from the keyboard, transforms it, and writes the transformation on the display screen. This behavior is called evaluation. Like the interpreter of a language, the Interpreter is constantly trying to determine what you mean by what you say. And just as the same word can mean different things in different contexts, the "meaning" of data to the Interpreter depends on information it has received in the past.

The Past, however, does not go back indefinitely; for The Interpreter, time begins with the first characters entered on the keyboard. Some things, therefore, must have meaning by themselves. Numeric Atoms have this property; when the Interpreter reads a number from the keyboard, its "translation" of that number is the number itself. We say

NUMBERS EVALUATE TO THEMSELVES.

The symbol "-=>" expresses this notion. The statment "5 evaluates to 5" is written "5 -=> 5". So according to the rule:

```
   0 -=> 0.
-512 -=> -512.
  33 -=> 33.
```

To get a feel for the Interpreter as a program and as a manipulator of data, we describe how the evaluation of numbers happens. The Interpreter is a collection of small programs, called procedures, which communicate with each other by passing data cells back and forth. The most important procedures are PRINT, EVALuate, and READ. PRINT is always looking for things to type on the display screen. When there is nothing to type, PRINT askes READ to get something. READ looks at the characters coming in from the keyboard. If it finds a digit, it obtains a new cell and starts building a Numeric Atom. As long as digits keep coming, READ keeps building. When the Atom has been built, READ gives it to PRINT. Because everyting is evaluated, PRINT hands the cell directly to EVAL, for examination and possibly transformation. EVAL inspects the cell, first asking if it is the empty Fern, <>. It is not. EVAL then asks if the cell is an Atom; in this example the cell is a Numeric so the answer is yes. Because it is a Numeric and not a Literal, EVAL gives the same cell back to PRINT. Now that the data is evaluated, PRINT is free to write its value on the display. Seeing that it is a Numeric, PRINT extracts the number from the contents of the cell, and prints it out. Having nothing more to do, PRINT asks READ for more data.

The Interpreter responds to Numeric input by returning a Numeric value. It behaves differently with Literal Atoms. When EVAL discovers that its Atomic argument is not a Numeric, it tries to find a meaning for the Atom in the

Environment, which is something like a dictionary. The Environment is small when the Interpreter starts; it contains a meaning for only two Literals:

FALSE -=> (), and
TRUE -=> TRUE.

As it recieves more information from the user, the Interpreter enlarges the Environment. We describe how this happens in more detail later. Since Numerics have intrinsic value, the rest of the examples in this section will use them.

When a List is EVALuated, the Interpreter EVALuates each of its elements and returns a List of the values:

<1 2 3> -=> (1 2 3).

When the READ procedure sees the character "<", it builds a structure with List cells, and notifies the other procedures that the structure's elements are to be EVALuated. The result is printed using parentheses instead of brackets. The difference between brackets and parentheses is discussed later. For now, assume that brackets are how people write Lists, and that parentheses are how machines write Lists. It's like an accent. Parenthesized Lists are called Pure data. Here are a few more examples:

```
<> -=> ().
<1 <2 <3 <4>>>> -=> (1 (2 (3 (4)))).
<-1 <<<-1 0>> 1>> -=> (-1 (((-1 0)) 1)).
```

As a review, examine the List (1 2 (3 4) ()) in its cellular representation.

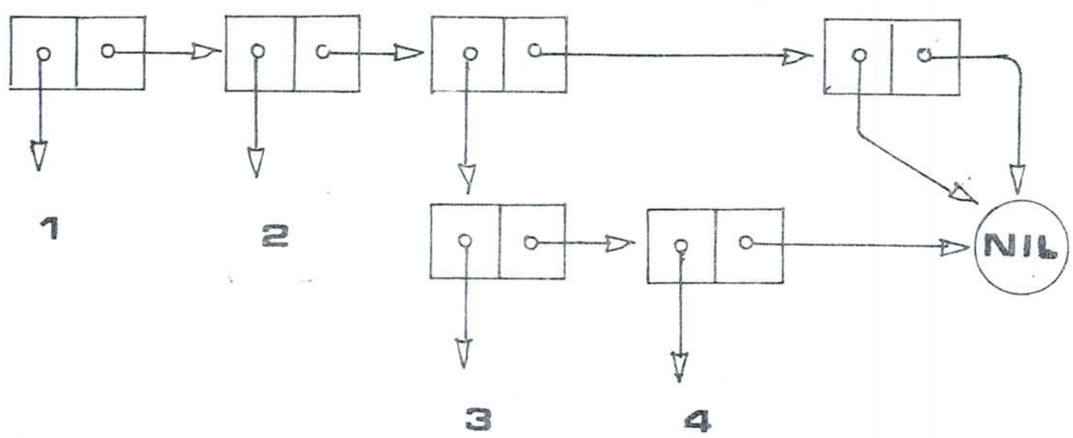


Figure A.3-1.

Inspect each cell in Figure A.3-1, and determine the role each plays in the structure.

A special character provided by the Interpreter permits the user to bypass the automatic evaluation of all input. If a Literal Atom is preceeded by a double-quote mark ("), the EVALuator returns the Literal as a value:

```
"EARTH -=> EARTH

"5 -=> 5

"DATA -=> DATA
```

List structures can also be quoted. When they are they must be expressed as Pure Data:

```
"(1 2 3) -=> (1 2 3)

"((1) ATOM) -=> ((1) ATOM)
```

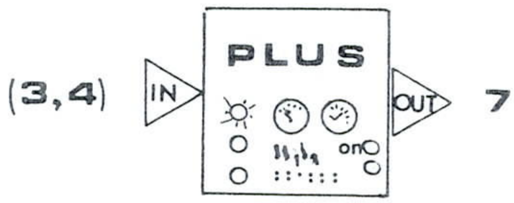
REVIEW OF SECTION A.3

1. The Interpreter is a READ-EVALuate-PRINT program.
   a. The PRINT procedure gets input data from READ, sends it to EVAL to be transformed into output data, and prints the structure EVAL sends back.
   b. Numeric Atoms evaluate to themselves.
   c. TRUE -=> TRUE; FALSE-=> ().
   d. If the EVALuator is given a Literal Atom, it searches the Environment for its meaning.

2. List structures are evaluated element by element.
   a. User structures are enclosed in brackets, Pure data structures are enclosed in parentheses.
   b. <> -=> ().

Section A.4 Functions

One way to describe a mathematical function is to say that it is an abstract mechanism. This concept can be depicted as a box with an "in" slot and an "out" slot. A valid argument is presented to the in slot; the box produces a result according to some rule, and sends it through the out slot:



This description is even more fitting for the Interpreter; the mechanism is a physical entity, a computer. The difference between the mathematical notion of "function" and the Interpreter mechanism, "a function," is that the abstract function is an ideal machine which requires no fuel or raw materials to create its answer, while Interpreter functions need both time and space to produce their results. The difference may be of little concern to the programmer at first, but eventually the fact that ideas expressed through the machine are constrained by physical and temporal boundaries becomes more important.

There are two kinds of functions: system-defined and user-defined. System-defined functions are analogous to Numeric Atoms; the Interpreter knows how to execute them from the beginning. User defined functions are created by the user, by giving The Interpreter a demonstration of how they work. Since the realm of the Interpreter is data, the arguments and results of all functions are data.

In this section, we demonstate some of the system defined functions. At the same time, the reader will be learning the syntax of function invocation. Some of the functions shown here are arithmetic -- their arguments are numbers or lists of numbers, and their results are numbers; other examples are predicate functions -- whose results are truth values. The third kind of function, data manipulators, are demonstrated later.

Arithmetic functions act on Numeric Atoms and return Numeric results, but the values returned by by predicate functions are less intuitive. In a language oriented toward data, the notions of truth and falsity must be represented as data. In predicate terms, the Atoms FALSE and NIL, and the structures () and <> represent falsity. Everything else, including the Atom TRUE represent truth.

System Defined Functions

1. ADD1, SUB1

These functions take a single argument which must be a Numeric Atom. The result is a Numeric whose value is one more (ADD1) or one less (SUB1) than the argument.

Examples:

ADD1:2 -=> 3.
ADD1:-1 -=> 0.
SUB1:100 -=> 99.

2. PLUS, DIFF, TIMES, DIV, MOD

These are "binary" arithmetic functions. Each takes one argument, a List, whose first two elements are Numbers. The result is a Numeric whose value is the sum (PLUS), difference (DIFF), product (TIMES), integer quotient (DIV), or modulo reduction (MOD) of the List elements. If the argument to one of these functions has more than two elements, the third, fourth, and so on are ignored.

Examples:

PLUS:<2 2> -=> 4.
DIFF:<1 3 5 7> -=> -2.
TIMES:<4 2> -=> 8.
DIV:<4 3> -=> 1.
MOD:<4 3> -=> 3.

3. GREAT, LESS, SAME

These predicate functions compare numeric values. Like the binary arithmetic functions, the argument to any of these functions should be a List with at least two Numeric elements (SAME is more general, as shown below).

Examples:

GREAT:<203 200> -=> TRUE.
LESS:<5 -10> -=> ().
SAME:<6 6> -=> TRUE.

4. NOT (or NULL)

NOT returns TRUE if its argument is false, and () if it is not.

Examples:

NOT:<> -=> TRUE.
NOT:<1 2 3> -=> ().
NOT:<NIL> -=> ().

Composition of Functions

In the section on EVALuation, we showed how the Interpreter seeks a meaning for all input. We look at this behavior more closely here, showing what happens when function calls are EVALuated. We are building a vocabulary of *forms* which the Interpreter accepts as grammatically correct. Numeric Atoms are the simplest forms, and EVALuate to themselves. List forms are not much more difficult; they are sequences of forms enclosed in brackets. We have already seen some examples of a third kind of form: *Applicative Forms*, which have the syntax:

function : argument

The colon is the APPLY operator. When the procedure EVAL is given a structure marked with the APPLY operator, it first EVALuates the argument part of the structure. Then the function is APPLYed to the result according to the system- or user-definition. If the form PLUS:<2 2> is given to EVAL the following things happen:

1. EVAL recognizes that PLUS:<2 2> is an applicative form. It makes a note of the function name, PLUS, and EVALuates the argument.
2. The argument is <2 2>, a List form. Each element is evaluated, and the result is the Pure List (2 2).
3. EVAL retrieves the function it noted in step 1, and APPLYs it to the EVALuated argument.
4. The result, 4, is returned.

The fact that EVAL EVALuates the argument part of an applicative form first, enables the programmer to string a number of function calls together, that is, the argument part of an applicative form may itself be an applicative form. As an example, consider the form ADD1:PLUS:<2 2>.

1. EVAL recognizes that ADD1:PLUS:<2 2> is an applicative form. It notes the function name, ADD1, and begins to EVALuate the argument, PLUS:<2 2>.
2. As in the example above, the form is PLUS:<2 2>. EVAL notes the PLUS.
3. The argument, <2 2> is EVALuated, returning the pure List (2 2).
4. PLUS is APPLYd to the argument List, returning the Numeric 4.
5. EVAL discovers the ADD1 noted in step 1, and APPLYs it to the result, 4.
6. The value 5 is returned.

The stringing together of function calls like this is called *composition*. Of course, it is not necessary for the programmer to keep track of the details of how EVAL gets its answer; the essence of the process can be stated in a rule:

RULE: When an APPLICATIVE FORM,
function : argument
is EVALuated, the ARGUMENT is EVALuated first, and the FUNCTION is APPLYd to the result.

Examples:

(i) ADD1:ADD1:ADD1:5 -=> 8.

(ii) PLUS:< PLUS:<2 2> SUB1:0 > -=> 3.

(iii) NULL:PLUS:<2 ADD1:9> -=> ().

<u>Constructors and Probes</u>

There are six more system-defined functions: SAME (mentioned in the list of system defined functions), ATOM, FIRST, REST, CONS, and FCNS. These are the most important functions in the Interpreter; they are used to investigate and manipulate data structures. We wish to emphasize here that all functions, and especially these, accept a more general kind of data structure than Lists, namely Ferns. Recall that Ferns are like Lists, except that they do not have order.

ATOM and SAME are predicates and are used to examine and compare structures. ATOM returns TRUE exactly when its argument is an Atom. SAME's argument is a List whose first two elements are compared. TRUE is returned only when both elements are <>, or both are the same atom.

Examples:

(i) ATOM:TRUE -=> TRUE.

(ii) ATOM:5 -=> TRUE.

(iii) ATOM:<4> -=> ().

(iv) EQ:<FALSE FALSE> -=> TRUE.

(v) EQ:<5 5> -=> TRUE.

(vi) EQ:<<1 2 3> <1 2 3>> -=> NIL.

The functions FIRST and REST are Fern _probes_, used to explore structures. Each takes a Fern argument. FIRST's value is the first element of the Fern. If the argument is a List, this value is the lefthand arrow in the List cell. If the argument is an unordered Fern, it is FIRST's responsibility to _select_ a first element and return it. REST returns the Fern that remains when the first element is removed.

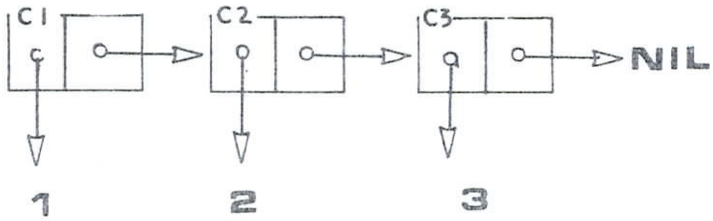
Examples:

(i) FIRST:<1 2 3> -=> 1.

(ii) FIRST:<PLUS:<2 2>> -=> 4.

(iii) FIRST:REST:<1 <2 <3>>> -=> (2 (3)).

Consider the form FIRST:<1 2 3>. The argument EVALuates to the Pure List (1 2 3):



Cell C1, a List cell, represents the argument. FIRST returns the lefthand pointer of C1, the Numeric 1. If REST is called on the same List, C1's right hand pointer is returned. It points to the List cell, C2, and hence to the rest of the the List, (2 3).

The next system-defined function is CONS, the constructor, used to build Lists. CONS's argument has two elements, the first may be any structure, the second must be a List (Fern). CONS returns a new Fern, whose first element is the first argument-element. The rest of the new Fern is the second argument-element.

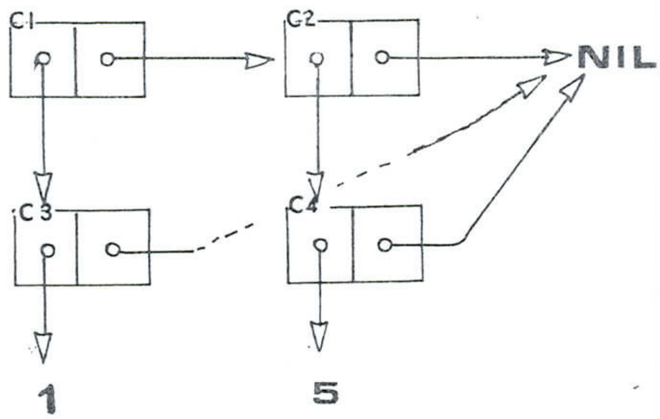
Examples:

(i) CONS:<1 <2 3>> -=> (1 2 3).

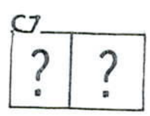
(ii) CONS:<<1> <>> -=> ((1)).

(iii) CONS:<2 CONS:<4 <6 8>>> -=> (2 4 6 8)

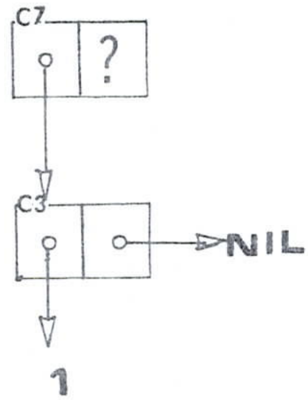
Let's do one more example, this time looking at the cells. Suppose the form is CONS:<<1> <5>>. EVALuated, the argument is the Pure List ((1) (5)):



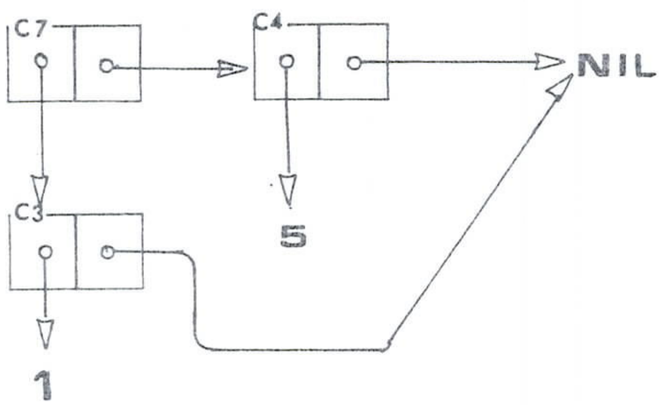
Cell C1 represents this List, which has two elements, C3 and C4. These are the elements that CONS uses. CONS returns a newly acquired cell, C7, but what are its arrows?



The left arrow points to the first argument-element, C3.



The right arrow points to the cell C4:



So CONS returns the Pure List ((1) 5).

Take the time to evaluate the following forms by hand. By doing so, you will be well on your way to mastering both the skill of functional composition, and the data-oriented behavior of the Interpreter. Use the cell representation to do the evaluation, but convert the result to List form.

1. FIRST:<<1 2> 3>.

2. REST:FIRST:<<1 2> 3>.

3. FIRST:REST:<<1 2> 3>.

4. REST:CONS:<TRUE <TRUE <>>>.

5. FIRST:CONS:<TRUE <22 33>>.

6. CONS:<FIRST:<1 2 3> REST:<1 2 3>>.

7. EQ:<CONS:<FIRST:<1> <>> <1>>.

ANSWERS:

1. (1 2)

2. (2)

3. 3

4. (TRUE ())

5. TRUE

6. (1 2 3)

7. ()

The last system defined function we shall mention is called FCNS. FCNS is a relative of CONS; it is a constructor. Instead of building Lists, it creates structures called multisets which are more gerneral than Lists because they are not ordered. FCNS is discussed in more detail in Section A.6.

Numeric Functions

A Numeric Atom in the function part of an applicative form is a shorthand notation for a common composition of Fern probes. The value of the form is the element of the argument that corresponds to the Numeric.

Examples:

(i) 1:<1 2 3> -=> FIRST:<1 2 3> -=> 1.

(ii) 2:<1 2 3> -=> FIRST:REST:<1 2 3> -=> 2.

(iii) 3:<1 2 3> -=> FIRST:REST:REST:<1 2 3> -=> 3.

and so on...

(iv) 1:2:1:<<1 <2>> 3 4> -=> (2).

(v) 1:1:1:1:1:<<<<<TRUE>>>>> -=> TRUE.

User-defined Functions

The addition of User-defined functions to the Interpreter's vocabulary opens a universe of meaningful computation to the user. As we describe how this is done, we will also be explaining how Environments are created.

These two concepts go hand in hand, and instead of trying to express their interelationship all at once in a single rule, we present several examples and discuss them. The reader will develop a feel for what is going on by studying the examples ard and practicing with the Interpreter. Function definition is the last addition to the semantics of the Interpreter; once it is understood the reader becomes a fully competent programmer in the Interpreter's language.

Example One

```
DEFINE  ADD2  NUMBER
      ADD1:ADD1:NUMBER. -=> ADD2
ADD2:5 -=> 7
```

Discussion

We wish to define a function which acts like ADD1, but returns a value two greater than that of its Numeric argument. The Atom DEFINE is a signal to the Interpreter that a function definition follows. The next thing in the input is the <u>function name</u> which must be an Atom. In this example, the function name is ADD2. Next comes a pattern of the argument which ADD2 will expect. In this example, the Atom NUMBER is the pattern; it is called the <u>formal parameter</u>. Following the formal parameter is the <u>function body</u>, a statement of what the function does. A function definition is a kind of form:

DEFINE name formal-parameter body.

Except for a few special cases, function definitions are the last of the Interpreter's form types.

<u>FORMs</u>
Atoms
Ferns
Applicative forms
Definitional forms

When a difinitional form is presented to the Interpreter, the function body and the formal parameter are added to a section of the Environment, called the FLIST, under the function name. Later, if ADD2 is invoked, the EVALuator will look up the function in the FLIST. The formal parameter will be compared with the actual argument (after EVALuation). This is called <u>binding</u>. As a result of the comparison new meanings for Atoms in the formal parameter are added to the Environment, and under the influence of these new meanings, the function body is executed. Consider the form ADD2:5.

1. The evaluator recognizes that ADD2:5 is an applicative form. It notes the function name, ADD2, and EVALuates the argument, 5.
2. 5 is a Numeric, and EVAluates to 5.
3. The EVALuator discovers that the function is user-defined. It looks the definition up in the FLIST.
4. The formal parameter is the Atom NUMBER. NUMBER is bound to the argument, 5.
5. EVAL now executes the function body, whose form is ADD1:ADD1:NUMBER. It notes the function name, ADD1 and EVALuates the argument, ADD1:NUMBER.
6. ADD1:NUMBER is another applicative form. EVAL notes the function, ADD1, and EVALuates the argument, NUMBER.
7. NUMBER is an Atomic form, but not a number! So EVAL looks for a meaning for NUMBER in the Environment.

NUMBER's meaning is its binding in the environment (step 4) the Numeric, 5. 5 is returned as the (step 6) argument value.

8. The ADD1 noted in step 6 is APPLYd to 5, returning 6.
9. The ADD1 noted in step 5 is APPLYd to 6, returning 7.
10. 7 is returned as a value of the form ADD2:5.

We are now beginning to understand how the meanings of Literal Atoms are created. *Literals are bound when user-defined functions are invoked.*

Example Two

```
DEFINE SUMUP   LIST
      IF NULL:LIST THEN 0
    ELSE PLUS:< FIRST:LIST  SUMUP:REST:LIST>.
-=> SUMUP
SUMUP:<1 2 3 4 5> -=> 15.
```

Discussion

We wish to define a function whose argument is a List of Numeric Atoms. The value of the function is the sum of the values of the elements of the List. The formal parameter of SUMUP is an Atom, like ADD2's. The body of SUMUP has two striking features. It returns one value if the argument-List is empty, another if it is not. When LIST is not empty, PLUS is called; the elements of PLUS's argument List are two numbers: the first element of LIST and the SUMUP of the rest of LIST. Functions whose bodies contain a call to themselves are *recursive*.

Example Three

```
DEFINE MERGE   (LIST1 LIST2)
   IF NULL:LIST2 THEN LIST1
ELSEIF NULL:LIST1 THEN LIST2
  ELSE CONS:< FIRST:LIST1
              CONS:< FIRST:LIST2
                     MERGE:< REST:LIST1  REST:LIST2>>>.
 -=> MERGE
MERGE:<<1 2 3> <4 5 6 7>> -=> (1 4 2 5 3 6 7)
```

Discussion

The function MERGE takes two Lists and creates a third whose elements are those of the first two. It's definition is similar to SUMUP's; it returns alternative results, depending on the structure of its argument-elements, and it is recursive. But there are differences too. The result of MERGE is a List instead of a number. Most user-defined functions do structure manipulation and not arithmetic. MERGE has more alternatives than SUMUP. In general a function can have as many alternatives as it needs. MERGE's formal parameter is not Atomic. When the time comes for parameter bindings, the formal parameter is compared with the argument. It acts as a template; the formal parameter's structure is superimposed on that of the EVALuated argument; its Atomic elements are bound to the corresponding structures in the argument (for this reason formal parameters are expressed as Pure data). In the example, when MERGE:<<1 2 3> <4 5 6 7>> is called, LIST1 will be bound to (1 2 3) and List2 to (4 5 6 7) -- at first.

These examples give just a flavor of programming with the Interpreter. In subsequent sections, more involved examples help develop a taste. The next section records a sample session with the Interpreter; the reader should follow this session at a terminal, interacting with the program. After that, try the problems on the next page.

PROBLEMS

1. Define a function whose argument has two elements, a number and a List of numbers. The number is the length of the List. The value of the function is the average of the numbers in the List
2. Define a function whose argument is a List of numbers. The value of the function is the average of the numbers in the List.
3. Finish this difinition:

```
    DEFINE BALANCE (ACCOUNT TRANSACT)
              IF NULL:TRANSACT THEN BALANCE
              .........
```

   The argument for BALANCE has two elements, ACCOUNT is a checkbook balance. TRANSACT is a List of transactions. Each element of TRANSACT is a List of the form

```
    <"CHECK  n> or <"DEPOSIT n>,
```

   where 'n' is a number. The purpose of the the functions is to subtract the amount of checks, and add the deposits to ACCOUNT. for example:

```
    BALANCE:< 50
                 <<"CHECK 10>
                  <"DEPOSIT 12>
                  <"CHECK 16>>> -=> 36.
```

REVIEW OF SECTION A.4

1. The EVALuator is a function whose argument has two elements: a form and an environment.
   a. If the form is a Numeric Atom, that Atom is returned; if it is a Literal, the Environment is searched for its meaning.
   b. If the form is a List, each of its arguments is EVALuated and a List of results is returned.
   c. If the form is applicative, the argument is evaluated first, then the function is APPLYd to the result.
   d. If the form is definitional, the function definition is added to the FLIST.

2. Forms have the following syntax:
   a. Atoms -- Numerics or Literals
   b. List -- <form form form...>
   c. Applicative forms --
      function : argument
   d. Definitional forms --
      DEFINE name formal-parameter body .

3. The Interpreter has system-defined functions for arithmetic and data manipulation.
   a. Unary arithmetic functions -- ADD1, SUB1.
   b. Binary arithmetic functions -- PLUS, DIFF, TIMES, DIV, MOD.
   c. Binary comparison predicates -- GREAT, LESS, SAME.
   d. Truth predicate -- NOT.
   e. Data inspection predicates -- NULL, NOT, SAME.
   f. Data probes -- FIRST, REST, and positive Numeric Atoms.
   g. Data constructors -- CONS, FONS.

4. The formal parameter part of a user function definition is used to bind Literal Atoms to their meanings during function execution.

5. Function bodies are a statement of the behavior of the function. They may contain a single form, or alternative forms, in which case the meaning of the function depends on conditional predicates.

Section A.5 A Sample session

In this section, an interactive session with the Interpreter is shown, starting with the Indiana University log-on procedure.

```
 77/08/24. 13.05.12.
INDIANA UNIVERSITY - LEVEL 9.           KRONOS 2.1-397/397
USER NUMBER:  3397b,sdj,
                            TERMINAL:      52,TTY
RECOVER /SYSTEM:full
                         READY.

batch              Get into BATCH mode.
/set,interp        Get the Interpreter.
/rfl,30000         Reserve some core space
/interp            Execute the Interpreter.


 ---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT
? 7000                                  ; Always type 7000 here.

?                                       ; The semicolon is a comment
?                                       ; character.  When the Interpreter
?                                       ; sees it, it skips to the next
?                                       ; line of input.  The Interpreter
?                                       ; delivers a prompt, "?" when it
?                                       ; wants more input.
? true
?                    ; Here, the Interpreter is looking ahead for

 -=>TRUE
?                    ; the APPLY operator.  To avoid extraneous
?                    ; prompts, terminate all forms with a period.
? true.

 -=>TRUE
? nil.

 -=>()
? false.

 -=>()
?
```

```
                           ; TRUE evaluates to TRUE; FALSE and NIL evaluate
?                          ; to the empty list.  No other Literal Atoms
?                          ; have meaning at the top level.
? someatom.

-=>-=> EVALUATION ERROR:  UNBOUND VARIABLE,
 SOMEATOM
-=>#BOTTOM#
?                          ; The interpreter was asked to evaluate an
?                          ; unbound Literal.  When errors occur, the "value"
?                          ; #BOTTOM# is returned as an answer.
?                          ;
? 1001.                    ; Numeric Atoms evaluate to themselves.

 -=>1001
? 10. 20. 30. 40.          ; Several forms may appear on the same line.

 -=>10

 -=>20

 -=>30

 -=>40
? <>.                      ; Lists are evaluated element-by-element.

 -=>()
? <1 2 3 4 5>.

 -=> (1 2 3 4 5)
? <<<1 2> 3 4> 5 <>>.

 -=> (((1 2) 3 4) 5 ())
?                          ;
?                          ;
? "someatom                ; The QUOTE character supresses automatic evaluation.

 -=>SOMEATOM
? "(((mean (mister mustard)) ()) )    ;QUOTEd lists should be not pure.

 -=> (((MEAN (MISTER MUSTARD)) ()))
?                          ;
?                          ; Numerics which are too large are reduced;
?                          ; literals which are too long are truncated.
? 1 2 3 4 5 6 7 8 9 0   opps! a mistake.  the ESCape key rejects the line
1234567890
<=-
LITERAL OR NUMERIC EXCEEDS BOUNDS.
? .                        ; 1234567890 evaluates to the biggest number

 -=>65535
? "excessivelylargeatom          ;the Interpreter can handle.
<=-
LITERAL OR NUMERIC EXCEEDS BOUNDS.

 -=>EXCESSIV
?
```

```
                              ; Here are some of the system-defined functions:
? add1:1.  sub1:0.

 -=>2

 -=>-1
? plus:<32 73>. times:<37 83>.  diff<4 27>. mod:<37 490>.

 -=>105

 -=>3071

-=>-=> EVALUATION ERROR:  UNBOUND VARIABLE,
 DIFF
-=>#BOTTOM#

 -=> (4 27)

 -=>9
?                         ; Another mistake. I forgot the APPLY operator
? diff:<4 27>             ; in the form DIFF<4 27>.

 -=>-23
? great:<5 8>.  less:<4 47>.  same:<39 39>.

 -=>()

 -=>TRUE

 -=>TRUE
? plus:<2 2 2 2>.         ; Binary system functions can be given too many

 -=>4
? plus:<true false>.      ; argument-elements.  The argument-elts. must be

-=>-=> EVALUATION ERROR:  NON-NUMERIC ARGUMENT,
 TRUE
-=>#BOTTOM#
?                         ; Numeric atoms in arithmetic functions.
?                         ; Functional composition.
? add1:sub1:add1:12.

 -=>13
? plus:<add1:add1:0 times:<3  div:<3 2>>>.

 -=>5
?                         ; The function part of a applicative form
?                         ; may be a List (fern).  The argument is then
?                         ; taken to be an array and the function-elements
?                         ; are APPLY'd to its columns.
? <plus plus>:<
? < 10   20 >
? <  1    2 > >.

 -=> (11 22)
?
```

```
                              ; function elements can be Lists.  The
?                             ; argument must reflect the nesting.
? <<plus great> times>:<
? << 20   300 >   2  >
? << 30    5  >   3  > >.

 -=> ((50 TRUE) 6)
?                              ; Sometimes there is a need to leave
?                              ; some positions of the argument array blank.
?                              ; The hashmark is used for this purpose.
? <same null great>:<
? <true  #     5  >
? <true  <>    4  > >.

 -=> (TRUE () TRUE)
? <1 2 3 <4 5] 6 >             ; bracketed expressions must be balanced.

-=>-=>SYNTAX ERROR: UNBALANCED SQUARE-BRACKET.
? average:<4 7>.               ; The user can define functions.

-=>-=> EVALUATION ERROR:  UNDEFINED FUNCTION,
 AVERAGE
-=>#BOTTOM#
? define AVERAGE (x y)  div:<plus:<x y> 2>.

 -=>AVERAGE
? average:<4 7>.

 -=>5
?                               ; Let's make AVERAGE more general.
? define AVERAGE  list
?         div:<sumup:list  length:list>.

 -=> (AVERAGE REDEF)
? define SUMUP list
?         if null:list then 0
?      else plus:<first:list  sumup:rest:list>.

 -=>SUMUP
? define LENGTH  list
?         if null:list then 0
?      else add1:length:rest:list.

 -=>LENGTH
? average:<4 7>.

 -=>5
? average<4 7 8 3 0 100 3000 4 66 7 3 4 7 99 123  Forgot APPLY  *DEL*
average:<4 7 8 3 0 100 3000 4 66 7 3 4 7 99 123>.

 -=>229
?
```

```
                        ; Let's look now at CONS.
? cons:<1 <2 3>>.

 -=> (1 2 3)
?                       ;  When the user ntices a mistake too late
?                       ;  to use the ESCape key, a double slash
?                       ;  causes the Interpreter to cancel the
?                       ;  form it is building and start again.
? define Joooin (list1 lits2)     ;  I wanted zzzzzzzz  "Join".
?       if null:lists then <>
?     else   //
?  define Join (list1 list2)
?       if null:list1 then <>
?     else cons:< first:list1  join:<//  ;another mistake.
? define JOIN (list1 list2)
?       if null:list1 then list2
?     else cons:<first:list1  join:<rest:list1 list2>>.

 -=>JOIN
? join:<<1 2 3><4 5 6>>.

 -=> (1 2 3 4 5 6)
?                              ; Most interesting problems deal with
?                              ; data structures and not arithmetic.
? define TIPS  list
?       if atom:list then list
?   elseif atom:first:list then cons:<first:list  tips:rest:list>
?     else join:<tips:first:list  tips:rest:list>.

 -=>TIPS
? tips:<1 2 3 <<4 5> 6> <<<7>>> <<true > false> "all <<<<< "done>>>>>.

 -=> (1 2 3 4 5 6 7 TRUE () ALL DONE)
?                       ;
?                       ;
?                       ;  To leave the Interpreter, type "exit.".
? exit.

-=>-=>-=>-=> LEAVING.
NODES DISPOZED, 13188
NODES RECYCLED, 1773
AVAIL--> 1098
/
```

Section A.6 CONS Revisited

In this section we take another look at the function CONS. The behavior of the Interpreter's constructor functions is not as direct as we have implied, and while in most cases, the difference is not important to the user, the subtle change we describe here gives him more power than he could have assumed.

The EVALuator as a function requires an argument List with two elements, a form and an Environment. It gets the form from top level input, or from a function definition; it gets its Environment through the process of binding formal parameters to function arguments. As the EVALuator proceeds through the the definition of a function, the Environment grows and diminishes as more parameters are bound and as unneeded bindings are discarded.

We now propose a change in the description of the constructor. Instead of evaluating its argument-elements and putting arrows to the results in the new CONS-List cell, a new kind of data structure, called a <u>suspension</u> is created and the arrows point to it. A suspension contains just enough information to enable the EVALuator to find the right value if it needs to. The required information, as we have often said, is a form and an Environment.

Consider the function call

CONS:<ADD1:1 <PLUS:<2 2>>>

The job of the CONStructor is to create a new List whose FIRST element is the value of the form ADD1:1, and the REST of which is the List <PLUS:<2 2>>. If these argument-elements are fully EVALuated, the resulting new List is (2 4). But it is wasteful to do this EVALuation; we may never need the FIRST or the REST of the new List. For example, suppose the form is part of a larger applicative form:

NULL:CONS:<ADD1:1 <PLUS:<2 2>>>

The value of this form is (), that is, FALSE, regardless of the outcome of the EVALuation of CONS's arguments.

Now suppose that instead of doing the EVALuation, two Suspensions are created and placed in the new cell's fields (Figure A.6-1).

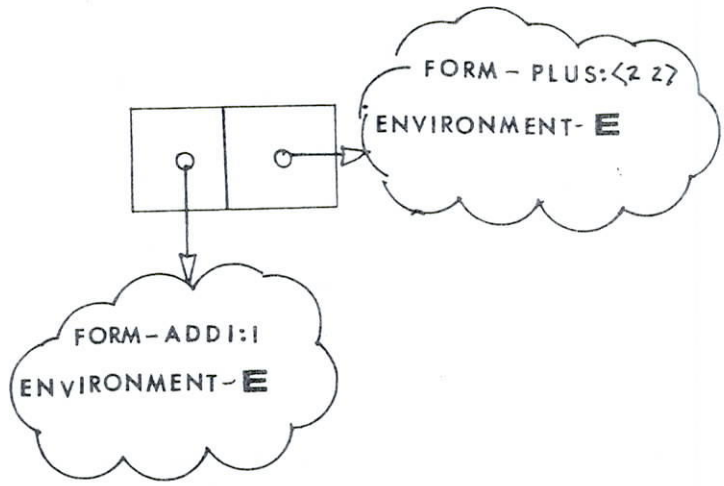


Figure A.6-1
Suspended construction.

The result is a List just as before, except that instead of being values, the FIRST and REST of the List are just promises: "Look at me and I will converge to a value." CONS no longer causes any computation to take place. If this structure is probed by the functions FIRST or REST, the Suspensions are coerced into values, as though they had been there all along.

FIRST and REST are no longer passive data examiners, dutifully returning the correct arrows, they are EVALuation drivers coaxing suspensions into values as they explore structures. If the form is:

FIRST:CONS:<ADD1:1 <PLUS:<2 2>>>,

1. The EVALuator notes the call to FIRST and EVALuates the argument, CONS:<...>.

2. No EVALuation takes place! CONS immediately builds a suspended List from the current Environment and the forms ADD1:1 and <PLUS:<2 2>>, (Figure A.6-1)
3. FIRST (step 1) is APPLYd to the result. The left hand suspension is found and coerced, that is ADD1:1 is EVALuated, returning the value 2. (N.B. this value replaces the suspension in the List cell so that no work is wasted by repeated calls to FIRST on the same cell)
4. The value 2 is returned.

In this example a probe of the suspended List structure yields the same result as would be returned if CONS had evaluated its arguments in the first place. However the evaluation of the REST of the List never takes place; about half of the work of construction is eliminated if that value is never referenced by a probe.

Now consider a definition:

```
DEFINE INTEGERS N  CONS:<N INTEGERS:ADD1:N>.
```

Assuming that the CONS in the definition evaluates its arguments, the form INTEGERS:1 is evaluated like this:

1. The form is applicative; the function INTEGERS is noted and the argument is evaluated.
2. 1 -=> 1.
3. INTEGERS is looked up on the FLIST; the Atom N is bound to 1; the form CONS:<N INTEGERS:ADD1:N>, is executed.
4. This too is an applicative form. CONS is noted and <N INTEGERS:ADD1:N> is evaluated.
5. This is a List form. Its first element is N, bound to 1 in step 3.

6. The second argument-element is the applicative form INTEGERS:ADD1:N. The function INTEGERS is noted and the argument is evaluated.

?!??!!!

No construction ever takes place. The Interpreter consumes all its time evaluating arguments to CONS and never executes CONS once. But if CONS suspends its arguements:

1. INTEGERS:1 an applicative form. The function INTEGERS is noted and the argument is evaluated.
2. 1 -=> 1.
3. the formal paramenter, N is bound to 1, and the body of INTEGERS is executed.
4. No evaluation takes place. The argument forms N and INTEGERS:ADD1:N, along with the environment binding N to 1, are turned into suspensions. a new cell is retrieved, its arrows made to point to the suspensions.
5. The new List cell is returned (Figure A.6-2).

If FIRST probes this structure it finds the suspended form N which evaluates to 1. If REST probes the structure, the form INTEGERS:ADD1:N is evaluated, returning a List of integers, starting with two.

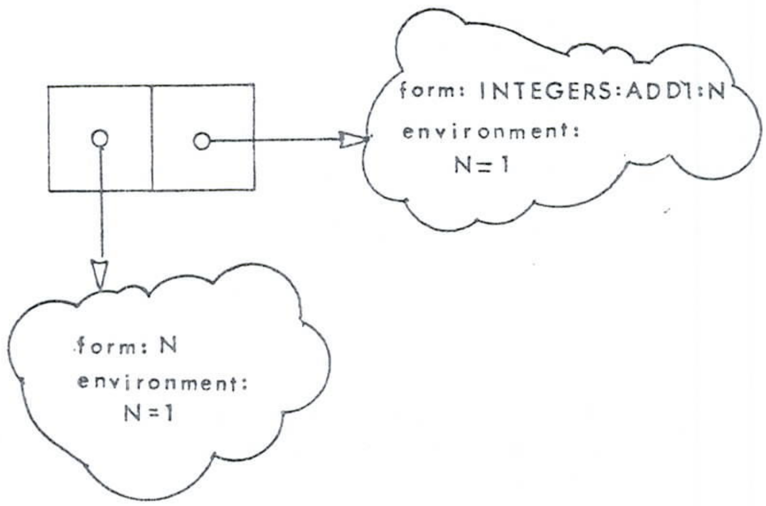


Figure A.6-2
The suspended List INTEGERS:1.

Suspensions are introduced to save the EVALuator from doing needless work. It is possible to build structures whose elements are partially computed, even though the user can never see them; if she probes the structure to look, the suspension is automatically coerced to a value. We introduce a second constructor, FONS, which goes even farther. FONS creates structures just as CONS, fetching a new memory cell, and filling its fields with suspensions. But unlike Lists (CONSed structures), Multisets (FONSed structures) are not ordered when they are built. If the argument to a probe is a Multiset, all suspensions in the Multiset are EVALuated simultaneuosly. The first actual value returned is declared the first element in the Multiset. The form

FONS:<1 FONS:<2 FONS:<3 <>>>>,

when fully coerced may evaluate to any of the following Pure Lists:

```
(1 2 3)  (1 3 2)  (2 1 3)
(2 3 1)  (3 1 2)  (3 2 1).
```

depending on the order in which suspensions return values. The eventual order of a Multiset can and should not be predictable for the user; if order is needed, use CONS.

The user's brackets are a shorthand notation for repeated calls to CONS:

```
<1 2 3> -=> CONS:<1 CONS:<2 CONS:<3 <>>>>.
```

Square brackets are used to build Multisets:

```
[1 2 3] -=> FONS:<1 FONS:<2 FONS:<3 [ ]>>>.
```

In this section we have introduced elements of the Interpreter's semantics, almost as an afterthought. In fact, the notion that computation is not carried out by the structure <u>builder</u>, but through exploration of the structure by <u>probes</u>, is one reason why the interpreter was written; few programming languages exist which explore this approach to computation.

Section A.7 A Sample program

Programs can be run non-interactively with the Interpreter.

Here is an example.

```
 77/08/24. 12.34.45.
INDIANA UNIVERSITY - LEVEL 9.          KRONOS 2.1-397/397
USER NUMBER:  3397b,sdj,
                               TERMINAL:     52,TTY
RECOVER /SYSTEM:full
                          READY.

batch                     Get into BATCH mode.
/get,interp               Get the Interpreter.
/get,maze2                Get the source program.
/rfl,30000                Reserve some core space.
/interp,maze2             Execute the Interpreter with source as input.


 ---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT

 -=> (TRACE 1)

   ;  THIS IS A PROGRAM TO FIND A PATH THROUGH A 3X3 MAZE.
   ;  THE MAZE IS A MATRIX OF TRUTH VALUES: ORDERED BY ROWS.
   ;  THE VALUE "TRUE" INDICATES THAT THE PATH TO THE GOAL CANNOT
   ;  GO THROUGH THAT POINT.  THE TOP-LEVEL FUNCTION, "MAZE"
   ;  IS GIVEN A GOAL POSITION AND A LIST OF POSSIBLE PATHS.
   ;  FOR EACH PATH IN THE LIST "MAZE" CHECKS TO SEE IF THE MOST
   ;  RECENT POSITION IS THE GOAL; IF IT IS, THE PATH IS SUCESSFULL
   ;  AND "MAZE" RETURNS IT AS AN ANSWER.  OTHERWISE, THAT PATH IS
   ;  IS REPLACED BY FOUR PATHS, ONE FOR EACH MOVE.  PATHS WHICH
   ;  END IN FAILURE ARE ELIMINATED ALONG THE WAY.
   ;
   ;
   ;
   ;
   ;
   ;
 DEFINE MAZE (GOAL PATHS)
       IF NULL:PATHS THEN <"NO "PATH "TO "GOAL>
   ELSEIF NULL:FIRST:PATHS THEN MAZE:<GOAL REST:PATHS>
   ELSEIF SAMESPOT:<GOAL 1:1:1:PATHS> THEN 1:1:PATHS
     ELSE MAZE:<GOAL
              APPEND:<MOVES:FIRST:PATHS  REST:PATHS>>.
  -=>MAZE
```

```
DEFINE APPEND (LIST1 LIST2)
      IF NULL:LIST1 THEN LIST2
    ELSE CONS:<1:LIST1  APPEND:<REST:LIST1 LIST2>>.
 -=>APPEND

DEFINE SAMESPOT ((X1 Y1) (X2 Y2))  ;  SEE IF TWO POSITIONS ARE THE SAME
      IF SAME:<X1 X2> THEN SAME:<Y1 Y2>
    ELSE FALSE.
 -=>SAMESPOT

DEFINE MOVES (PATH STATE)  ;TRY A MOVE IN EACH DIRECTION.
    <  MOVE:<<0 1> STATE PATH>    ; MOVE UP,
       MOVE:<<0 -1> STATE PATH>   ; DOWN,
       MOVE:<<-1 0> STATE PATH>   ; LEFT,
       MOVE:<<1 0> STATE PATH>  >.
 -=>MOVES

DEFINE MOVE (DIRECTION STATE PATH)
      ;  "MOVE," "MOVE1," AND "MOVE2" ATTEMPT TO
      ;  ADD ANOTHER POSITION TO A PATH.
      ;   IF THE NEW POSITION IS MARKED TRUE,
      ;  THE PATH IS ELIMINATED FROM CONSIDERATION.
      MOVE1:<DRECTION STATE 1:PATH PATH>.
 -=>MOVE

DEFINE MOVE1 ((DX DY) STATE (PX PY) PATH)
      MOVE2:<<PLUS:<DX PX> PLUS:<DY PY>> STATE PATH>.
 -=>MOVE1

DEFINE MOVE2 ((SX SY) STATE PATH)
      IF OR:< LESS:<SX 1>  LESS:<SY 1>
              GREAT:<SX 3>  GREAT:<SY 3>
              MARKED:<<SX SY> STATE>  >     THEN <>
    ELSE < CONS:<<SX SY> PATH> MARK:<STATE 1:PATH>>.
 -=>MOVE2

DEFINE OR LIST
      IF NULL:LIST THEN FALSE
  ELSEIF FIRST:LIST THEN TRUE
    ELSE OR:REST:LIST.
 -=>OR

DEFINE MARKED ((X Y) STATE);  SEE IF A POSITION HAS BEEN MARKED
      IF SAME:<Y 1> THEN MARKHELP:<X FIRST:STATE>
    ELSE MARKED:<<X SUB1:Y> REST:STATE>.
 -=>MARKED

DEFINE MARKHELP (X ROW)
      IF SAME:<X 1> THEN FIRST:ROW
    ELSE MARKHELP:<SUB1:X REST:ROW>.
 -=>MARKHELP

DEFINE MARK (STATE (X Y))  ;  MARK A POSITION.
      IF SAME:<Y 1> THEN CONS:<MARKROW:<FIRST:STATE X> REST:STATE>
    ELSE CONS:<FIRST:STATE MARK:<REST:STATE <X SUB1:Y>>>.
 -=>MARK
```

```
DEFINE MARKROW ((R1 R2 R3) X)
      IF SAME:<X 1> THEN <TRUE R2 R3>
  ELSEIF SAME:<X 2> THEN <R1 TRUE R3>
    ELSE <R1 R2 TRUE>.
 -=>MARKROW

   ;
   ;
   ;  WE'LL CALL 'MAZE' WITH A GOAL POSITION OF <1 1> -- THE
   ;  UPPER-LEFT CORNER -- AND A STARTING POSITION OF <3 3>.
   ;  PATHS THROUGH THE THE POINTS (3,1), (2,1), (2,3), AND
   ;  (1,3) ARE BLOCKED.
MAZE:<<1 1> <<<<3 3>> <<FALSE  FALSE  TRUE >
                        <TRUE   FALSE  TRUE >
                        <TRUE   FALSE  FALSE>>>>.
 -=> ((1 1) (2 1) (2 2) (2 3) (3 3))

   ;
   ;
   ;  NOW TRY TO FIND A PATH THAT DOESN'T EXIST.
MAZE: <<1 1>  <<<<3 3>>  <<FALSE  TRUE   FALSE >
                           <TRUE   FALSE  FALSE >
                           <FALSE  FALSE  FALSE >>  >> >.
 -=> (NO PATH TO GOAL)

EXIT.
-=>-=>-=>-=> LEAVING.
NODES DISPOZED, 138439
NODES RECYCLED, 13627
AVAIL--> 1687
/
```

Here is a copy of the source file. The first two lines should be duplicated in all programs.

```
rewind,maze2
/copy,maze2
7000
(TRACE 0)
   ;  THIS IS A PROGRAM TO FIND A PATH THROUGH A 3X3 MAZE.
   ;  THE MAZE IS A MATRIX OF TRUTH VALUES; ORDERED BY ROWS.
   ;  THE VALUE "TRUE" INDICATES THAT THE PATH TO THE GOAL CANNOT
   ;  GO THROUGH THAT POINT.  THE TOP-LEVEL FUNCTION, "MAZE"
   ;  IS GIVEN A GOAL POSITION AND A LIST OF PO
 *TERMINATED*

/
```

The special form '(trace 1)' informs the Interpreter that the input is to be echoed to the output file. Output can also be specified in the execute command:

```
set,samples
/interp,samples,list          Specify both input and output.
/rewind,list                  To get a copy of the results....
/copy,list
 ---=>---=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT


 -=> (TRACE 1)

ADD1:1                                                        ;
 -=>2

PLUS:<2 2>                                                    ;
 -=>4


 *TERMINATED*

/
```

Finally, the Interpreter can be called from a procedure file; thus it can be executed by submitting a job whose control cards are the same as in this example.

Appendix B

Program Traces.

Several levels of tracing are exhibited here for runs of the same program on the interpreter. Level 1 is the normal trace of a submitted job; input is echoed to the output file. As the value of trace increases, the behavior of the evaluator is more explicitly shown.

```
 ---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT~


 -=> (TRACE 1)

FIRST:<1000 2000 3000>.
 -=>1000

EXIT.
-=>-=>-=>-=> LEAVING.
NODES DISPOSED, 13
NODES RECYCLED, 3
AVAIL--> 286



15.31.44.  SDJZBVN   END OF LISTING
```

```
 ---=>---=>-=> ENTERING VERSION 0.1          TRACE=3

-=> MEMORY LIMIT"


 -=> (
-=>-=>-=> IN CAR LOOP. CAR OF P IS 145TRACE
TRACE
-=>-=>-=> IN CDR LOOP.   CDR OF P IS 276[C[?3][*275][*0]]

-=>-=>-=> IN CAR LOOP. CAR OF P IS 275[A[?2][?3][?0]]
(3)
-=>-=>-=> IN CDR LOOP.   CDR OF P IS 0[C[?0][*0][*0]]
)

FIRST:<1000 2000 3000>.
-=>-=>-=> ENTER READ LOOP.  Q IS 274[C[?0][*257][*277]]

 -=>1000

EXIT.
-=>-=>-=> ENTER READ LOOP.  Q IS 243
 EXIT

-=>-=>-=>-=> LEAVING.
NODES DISPOSED, 18
NODES RECYCLED, 3
AVAIL--> 286



15.32.18.  SDJZBVN   END OF LISTING
```

```
 ---=>---=>-=> ENTERING VERSION 0.1          5

-=> MEMORY LIMIT^


 -=> (CAR

-=>-=>-=> IN CAR LOOP. CAR OF P IS 145TRACE
TRACECDR

-=>-=>-=> IN CDR LOOP.   CDR OF P IS 276[C[?3]['275]['0]]
 CAR

-=>-=>-=> IN CAR LOOP. CAR OF P IS 275[A[?2][?5][?0]]
5CDR

-=>-=>-=> IN CDR LOOP.   CDR OF P IS 0[C[?0]['0]['0]]
)

FIRST:<1000 2000 3000>.
-=>-=>-=> ENTER READ LOOP.  Q IS 274[C[?0]['257]['277]]
TOP
TOP
EVLIS
CDR
EVLIS1
ANB
      APPLY 169 TO 292
CAR

 CONTEXT PUSH: 294 --> ['293]['292]['0]], PROCESS-CAR.
RESTORE
TOP
TOP

CONTEXT POP: ['293]['292]['0]]; FILL-CAR.
RESTORE
CAR

 -=>1000

EXIT.
-=>-=>-=> ENTER READ LOOP.  Q IS 243
 EXIT

-=>-=>-=>-=> LEAVING.
NODES DISPOSED, 18
NODES RECYCLED, 3
AVAIL--> 286


15.30.21.  SDJZBWA   END OF LISTING
```

```
 ---=>---=>-=> ENTERING VERSION 0.1      7

-=> MEMORY LIMIT^

            ASSIGNMENT: 274.
            ASSIGNMENT: 0.

 -=> (            ASSIGNMENT: 274.
            ASSIGNMENT: 0.

-=>-=>-=>PROCESS CREATED: [S['274][?7]['0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 277.

 POP:
          PLACE: 7, EXP: 274, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CAR
            ASSIGNMENT: 145.
            ASSIGNMENT: 145.
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.

-=>-=>-=> IN CAR LOOP. CAR OF P IS 145TRACE
TRACE            ASSIGNMENT: 0.

-=>-=>-=>PROCESS CREATED: [S['274][?9]['0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 277.

 POP:
          PLACE: 9, EXP: 274, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CDR
            ASSIGNMENT: 276.
            ASSIGNMENT: 276.
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.

-=>-=>-=> IN CDR LOOP.   CDR OF P IS 276[C[?3]['275]['0]]
             ASSIGNMENT: 276.
            ASSIGNMENT: 0.

-=>-=>-=>PROCESS CREATED: [S['276][?7]['0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 274.

 POP:
          PLACE: 7, EXP: 276, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CAR
            ASSIGNMENT: 275.
            ASSIGNMENT: 275.
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.

-=>-=>-=> IN CAR LOOP. CAR OF P IS 275[A[?2][?7][?0]]
(7)          ASSIGNMENT: 0.

-=>-=>-=>PROCESS CREATED: [S['276][?9]['0]]
```

```
-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 274.

 POP:
          PLACE: 9, EXP: 276, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CDR
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.

-=>-=>-=> IN CDR LOOP.   CDR OF P IS 0[C[?0][']0][']0]]
            ASSIGNMENT: 276.
)

FIRST:<1000 2000 3000>.
-=>-=>-=> ENTER READ LOOP.  Q IS 274[C[?0]['257]['277]]

-=>-=>-=>PROCESS CREATED: [S['274][?1]['0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 286.

 POP:
          PLACE: 1, EXP: 274, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
TOP
     PUSH-1: 286--> [S['169][?25]['0]]
     PUSH-1: 287--> [S['278][?1]['286]]

 POP:
          PLACE: 1, EXP: 278, ENV: 0, REVAL: 0, MODE: 37, STACK: 286
-=>TOP
     PUSH-1: 287--> [S['280][?71]['286]]

 POP:
          PLACE: 71, EXP: 280, ENV: 0, REVAL: 0, MODE: 37, STACK: 286
-=>EVLIS
     PUSH-1: 287--> [S['280][?72]['286]]
     PUSH-1: 274--> [S['280][?9]['287]]

 POP:
          PLACE: 9, EXP: 280, ENV: 0, REVAL: 0, MODE: 37, STACK: 287
-=>CDR
            ASSIGNMENT: 282.

 POP:
          PLACE: 72, EXP: 280, ENV: 0, REVAL: 282, MODE: 37, STACK: 286
-=>EVLIS1
            ASSIGNMENT: 292.

 POP:
          PLACE: 25, EXP: 169, ENV: 0, REVAL: 292, MODE: 37, STACK: 0
-=>ANB
            ASSIGNMENT: 169.
            ASSIGNMENT: 292.
      APPLY 169 TO 292
     PUSH-1: 286--> [S['292][?7]['0]]

 POP:
          PLACE: 7, EXP: 292, ENV: 0, REVAL: 292, MODE: 37, STACK: 0
-=>CAR
```

```
     PUSH-1: 286--> [S['292][?7]['0]]
     PUSH-1: 293--> [S['0][?80]['286]]

 CONTEXT PUSH: 294 --> ['293]['292]['0]], PROCESS-CAR.

 POP:
          PLACE: 80, EXP: 0, ENV: 0, REVAL: 292, MODE: 19, STACK: 288
-=>RESTORE
            ASSIGNMENT: 0.

 POP:
          PLACE: 1, EXP: 287, ENV: 0, REVAL: 292, MODE: 19, STACK: 0
-=>TOP
     PUSH-1: 288--> [S['279][?1]['0]]

 POP:
          PLACE: 1, EXP: 279, ENV: 0, REVAL: 292, MODE: 19, STACK: 0
-=>TOP
            ASSIGNMENT: 279.

CONTEXT POP: ['293]['292]['0]]; FILL-CAR.
            ASSIGNMENT: 293.

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 293.

 POP:
          PLACE: 80, EXP: 0, ENV: 0, REVAL: 0, MODE: 37, STACK: 286
RESTORE
            ASSIGNMENT: 0.

 POP:
          PLACE: 7, EXP: 292, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
-=>CAR
            ASSIGNMENT: 279.
            ASSIGNMENT: 279.
            ASSIGNMENT: 0.

 -=>1000

EXIT.
-=>-=>-=> ENTER READ LOOP.  Q IS 243
 EXIT
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.

-=>-=>-=>-=> LEAVING.
NODES DISPOSED, 18
NODES RECYCLED, 3
AVAIL--> 286


15.32.54.  SDJZ3VN   END OF LISTING
```

9

```
---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT"

           ASSIGNMENT: 274.
           ASSIGNMENT: 0.

 -=> (            ASSIGNMENT: 274.
           ASSIGNMENT: 0.

-=>-=>-=>PROCESS CREATED: [S['274][?7]['0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 277.

 POP:
         PLACE: 7, EXP: 274, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CAR
           ASSIGNMENT: 145.
           ASSIGNMENT: 145.
           ASSIGNMENT: 0.
           ASSIGNMENT: 0.

-=>-=>-=> IN CAR LOOP. CAR OF P IS 145TRACE
TRACE             ASSIGNMENT: 0.

-=>-=>-=>PROCESS CREATED: [S['274][?9]['0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 277.

 POP:
         PLACE: 9, EXP: 274, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CDR
           ASSIGNMENT: 276.
           ASSIGNMENT: 276.
           ASSIGNMENT: 0.
           ASSIGNMENT: 0.

-=>-=>-=> IN CDR LOOP.   CDR OF P IS 276[C[?3]['275]['0]]
            ASSIGNMENT: 276.
           ASSIGNMENT: 0.

-=>-=>-=>PROCESS CREATED: [S['276][?7]['0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 274.

 POP:
         PLACE: 7, EXP: 276, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CAR
           ASSIGNMENT: 275.
           ASSIGNMENT: 275.
           ASSIGNMENT: 0.
           ASSIGNMENT: 0.

-=>-=>-=> IN CAR LOOP. CAR OF P IS 275[A[?2][?9][?0]]
            ASSIGNMENT: 0.

-=>-=>-=>PROCESS CREATED: [S['276][?9]['0]]
```

```
-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 274.

 POP:
          PLACE: 9, EXP: 276, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CDR
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.

-=>-=>-=> IN CDR LOOP.   CDR OF P IS 0[C[?0]['0]['0]]
            ASSIGNMENT: 276.
)

FIRST:<1000 2000 3000>.
-=>-=>-=> ENTER READ LOOP.  Q IS 274[C[?0]['257]['277]]

-=>-=>-=>PROCESS CREATED: [S['274][?1]['0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 286.

 POP:
          PLACE: 1, EXP: 274, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
TOP
     PUSH-1: 286--> [S['169][?25]['0]]
     PUSH-1: 287--> [S['278][?1]['286]]

 POP:
          PLACE: 1, EXP: 278, ENV: 0, REVAL: 0, MODE: 37, STACK: 286
-=>TOP
     PUSH-1: 287--> [S['280][?71]['286]]

 POP:
          PLACE: 71, EXP: 280, ENV: 0, REVAL: 0, MODE: 37, STACK: 286
-=>EVLIS
     PUSH-1: 287--> [S['280][?72]['286]]
     PUSH-1: 274--> [S['280][?9]['287]]

 POP:
          PLACE: 9, EXP: 280, ENV: 0, REVAL: 0, MODE: 37, STACK: 287
-=>CDR
            ASSIGNMENT: 282.

 POP:
          PLACE: 72, EXP: 280, ENV: 0, REVAL: 282, MODE: 37, STACK: 286
-=>EVLIS1
            ASSIGNMENT: 292.

 POP:
          PLACE: 25, EXP: 169, ENV: 0, REVAL: 292, MODE: 37, STACK: 0
-=>ANB
            ASSIGNMENT: 169.
            ASSIGNMENT: 292.
      APPLY 169 TO 292
     PUSH-1: 286--> [S['292][?7]['0]]

 POP:
          PLACE: 7, EXP: 292, ENV: 0, REVAL: 292, MODE: 37, STACK: 0
-=>CAR
```

```
 POP:
          PLACE: 7, EXP: 276, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CAR
            ASSIGNMENT: 275.
            ASSIGNMENT: 275.
            ASSIGNMENT: 0.
RECYCLE 275:[A[?3][?11][?0]]
            ASSIGNMENT: 0.
RECYCLE 276:[C[?3]['275]['0]]

-=>-=>-=> IN CAR LOOP. CAR OF P IS 275[A[?2][?11][?0]]
11              ASSIGNMENT: 0.
RECYCLE 275:[A[?2][?11][?0]]

-=>-=>-=>PROCESS CREATED: [S['276][?9]['0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 274.

 POP:
          PLACE: 9, EXP: 276, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CDR
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.
RECYCLE 276:[C[?3]['275]['0]]

-=>-=>-=> IN CDR LOOP.   CDR OF P IS 0[C[?0]['0]['0]]
            ASSIGNMENT: 276.
)

FIRST:<1000 2000 3000>.
-=>-=>-=> ENTER READ LOOP.  Q IS 274[C[?0]['257]['277]]

-=>-=>-=>PROCESS CREATED: [S['274][?1]['0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 286.

 POP:
          PLACE: 1, EXP: 274, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
TOP
     PUSH-1: 286--> [S['169][?25]['0]]
     PUSH-1: 287--> [S['278][?1]['286]]

RECYCLE 274:[C[?1]['257]['277]]
--> AVAIL
 POP:
          PLACE: 1, EXP: 278, ENV: 0, REVAL: 0, MODE: 37, STACK: 286
-=>TOP
     PUSH-1: 287--> [S['280][?71]['286]]

RECYCLE 278:[C[?2]['149]['280]]
 POP:
          PLACE: 71, EXP: 280, ENV: 0, REVAL: 0, MODE: 37, STACK: 286
-=>EVLIS
     PUSH-1: 287--> [S['280][?72]['286]]
RECYCLE 257:
 ??:??
     PUSH-1: 274--> [S['280][?9]['287]]
```

```
RECYCLE 280:[C[?5][‘279][‘282]]
 POP:
          PLACE: 9, EXP: 280, ENV: 0, REVAL: 0, MODE: 37, STACK: 287
-=>CDR
            ASSIGNMENT: 282.

RECYCLE 280:[C[?4][‘279][‘282]]
 POP:
          PLACE: 72, EXP: 280, ENV: 0, REVAL: 282, MODE: 37, STACK: 286
-=>EVLIS1
            ASSIGNMENT: 292.
RECYCLE 282:[C[?4][‘281][‘284]]

RECYCLE 280:[C[?4][‘279][‘282]]
 POP:
          PLACE: 25, EXP: 169, ENV: 0, REVAL: 292, MODE: 37, STACK: 0
-=>ANB
            ASSIGNMENT: 169.
            ASSIGNMENT: 292.
      APPLY 169 TO 292
     PUSH-1: 286--> [S[‘292][?7][‘0]]

RECYCLE 169:FIRST
 POP:
          PLACE: 7, EXP: 292, ENV: 0, REVAL: 292, MODE: 37, STACK: 0
-=>CAR
     PUSH-1: 286--> [S[‘292][?7][‘0]]
     PUSH-1: 293--> [S[‘0][?30][‘286]]

 CONTEXT PUSH: 294 --> [‘293][‘292][‘0]], PROCESS-CAR.

RECYCLE 292:[C[?5][?18][‘291]]
 POP:
          PLACE: 80, EXP: 0, ENV: 0, REVAL: 292, MODE: 19, STACK: 288
-=>RESTORE
            ASSIGNMENT: 0.

 POP:
          PLACE: 1, EXP: 287, ENV: 0, REVAL: 292, MODE: 19, STACK: 0
-=>TOP
     PUSH-1: 288--> [S[‘279][?1][‘0]]

RECYCLE 287:[C[?1][‘163][‘280]]
--> AVAIL
 POP:
          PLACE: 1, EXP: 279, ENV: 0, REVAL: 292, MODE: 19, STACK: 0
-=>TOP
            ASSIGNMENT: 279.
RECYCLE 292:[C[?4][?18][‘291]]

CONTEXT POP: [‘293][‘292][‘0]]; FILL-CAR.
RECYCLE 292:[C[?3][‘279][‘291]]
            ASSIGNMENT: 293.
RECYCLE 279:[A[?4][?1000][?0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 293.

RECYCLE 279:[A[?3][?1000][?0]]
 POP:
          PLACE: 80, EXP: 0, ENV: 0, REVAL: 0, MODE: 37, STACK: 286
```

```
RESTORE
            ASSIGNMENT: 0.

 POP:
          PLACE: 7, EXP: 292, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
-=>CAR
            ASSIGNMENT: 279.
            ASSIGNMENT: 279.
            ASSIGNMENT: 0.
RECYCLE 279:[A[?4][?1000][?0]]
-=>1000RECYCLE 279:[A[?3][?1000][?0]]


EXIT.
-=>-=>-=> ENTER READ LOOP.  Q IS 243
 EXIT
            ASSIGNMENT: 0.
RECYCLE 243:EXIT
            ASSIGNMENT: 0.
RECYCLE 276:[C[?3]['275]['0]]

-=>-=>-=>-=> LEAVING.
NODES DISPOSED, 18
NODES RECYCLED, 3
AVAIL--> 286



15.33.21.  SDJZ3VN   END OF LISTING
```

17

```
 ---=>--=>-=> ENTERING VERSION 0.1

-=> MEMORY LIMIT^

            ASSIGNMENT: 274.
            ASSIGNMENT: 0.
RECYCLE 274:[C[?3][']145][']276]]

 -=> (             ASSIGNMENT: 274.
            ASSIGNMENT: 0.
RECYCLE 274:[C[?3][']145][']276]]

-=>-=>-=>PROCESS CREATED: [S[']274][?7][']0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 277.

RECYCLE 274:[C[?3][']145][']276]]
 POP:
          PLACE: 7, EXP: 274, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CAR
            ASSIGNMENT: 145.
            ASSIGNMENT: 145.
            ASSIGNMENT: 0.
RECYCLE 145:TRACE
            ASSIGNMENT: 0.
RECYCLE 274:[C[?2][']145][']276]]

-=>-=>-=> IN CAR LOOP. CAR OF P IS 145TRACE
TRACE             ASSIGNMENT: 0.
RECYCLE 145:TRACE

-=>-=>-=>PROCESS CREATED: [S[']274][?9][']0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 277.

 POP:
          PLACE: 9, EXP: 274, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CDR
            ASSIGNMENT: 276.
            ASSIGNMENT: 276.
            ASSIGNMENT: 0.
RECYCLE 276:[C[?4][']275][']0]]
            ASSIGNMENT: 0.
RECYCLE 274:[C[?2][']145][']276]]

-=>-=>-=> IN CDR LOOP.   CDR OF P IS 276[C[?3][']275][']0]]
             ASSIGNMENT: 276.
RECYCLE 274:[C[?1][']145][']276]]
--> AVAIL
            ASSIGNMENT: 0.
RECYCLE 276:[C[?3][']275][']0]]
RECYCLE 145:TRACE

-=>-=>-=>PROCESS CREATED: [S[']276][?7][']0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 274.
```

```
 POP:
          PLACE: 7, EXP: 276, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CAR
            ASSIGNMENT: 275.
            ASSIGNMENT: 275.
            ASSIGNMENT: 0.
RECYCLE 275:[A[?3][?17][?0]]
            ASSIGNMENT: 0.
RECYCLE 276:[C[?3]['275]['0]]

-=>-=>-=> IN CAR LOOP. CAR OF P IS 275[A[?2][?17][?0]]
(17)          ASSIGNMENT: 0.
RECYCLE 275:[A[?2][?17][?0]]

-=>-=>-=>PROCESS CREATED: [S['276][?9]['0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 274.

 POP:
          PLACE: 9, EXP: 276, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
CDR
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.
            ASSIGNMENT: 0.
RECYCLE 276:[C[?3]['275]['0]]

-=>-=>-=> IN CDR LOOP.   CDR OF P IS 0[C[?0]['0]['0]]
            ASSIGNMENT: 276.
)

FIRST:<1000 2000 3000>.
-=>-=>-=> ENTER READ LOOP.  Q IS 274[C[?0]['257]['277]]

-=>-=>-=>PROCESS CREATED: [S['274][?1]['0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 286.

 POP:
          PLACE: 1, EXP: 274, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
TOP
     PUSH-1: 286--> [S['169][?25]['0]]
     PUSH-1: 287--> [S['278][?1]['286]]

RECYCLE 274:[C[?1]['257]['277]]
--> AVAIL
 POP:
          PLACE: 1, EXP: 278, ENV: 0, REVAL: 0, MODE: 37, STACK: 286
-=>TOP
     PUSH-1: 287--> [S['280][?71]['286]]

RECYCLE 278:[C[?2]['149]['280]]
 POP:
          PLACE: 71, EXP: 280, ENV: 0, REVAL: 0, MODE: 37, STACK: 286
-=>EVLIS
     PUSH-1: 287--> [S['280][?72]['286]]
RECYCLE 257:
 ??:??
     PUSH-1: 274--> [S['280][?9]['287]]
```

```
RECYCLE 280:[C[?5][‘279][‘282]]
 POP:
           PLACE: 9, EXP: 280, ENV: 0, REVAL: 0, MODE: 37, STACK: 287
-=>CDR
             ASSIGNMENT: 282.

RECYCLE 280:[C[?4][‘279][‘282]]
 POP:
           PLACE: 72, EXP: 280, ENV: 0, REVAL: 282, MODE: 37, STACK: 286
-=>EVLIS1
            DOTPAIR--> 287--> [[][163][280]].
            DOTPAIR--> 274--> [[][151][282]].
            CONS[292]-->[C[?0][‘289][‘291]]
             ASSIGNMENT: 292.
RECYCLE 282:[C[?4][‘281][‘284]]

RECYCLE 280:[C[?4][‘279][‘282]]
 POP:
           PLACE: 25, EXP: 169, ENV: 0, REVAL: 292, MODE: 37, STACK: 0
-=>ANB
             ASSIGNMENT: 169.
             ASSIGNMENT: 292.
       APPLY 169 TO 292
      PUSH-1: 286--> [S[‘292][?7][‘0]]

RECYCLE 169:FIRST
 POP:
           PLACE: 7, EXP: 292, ENV: 0, REVAL: 292, MODE: 37, STACK: 0
-=>CAR
      PUSH-1: 286--> [S[‘292][?7][‘0]]
      PUSH-1: 293--> [S[‘0][?80][‘286]]

 CONTEXT PUSH: 294 --> [‘293][‘292][‘0]], PROCESS-CAR.

RECYCLE 292:[C[?5][?18][‘291]]
 POP:
           PLACE: 80, EXP: 0, ENV: 0, REVAL: 292, MODE: 19, STACK: 288
-=>RESTORE
             ASSIGNMENT: 0.

 POP:
           PLACE: 1, EXP: 287, ENV: 0, REVAL: 292, MODE: 19, STACK: 0
-=>TOP
      PUSH-1: 288--> [S[‘279][?1][‘0]]

RECYCLE 287:[C[?1][‘163][‘280]]
--> AVAIL
 POP:
           PLACE: 1, EXP: 279, ENV: 0, REVAL: 292, MODE: 19, STACK: 0
-=>TOP
             ASSIGNMENT: 279.
RECYCLE 292:[C[?4][?18][‘291]]

CONTEXT POP: [‘293][‘292][‘0]]; FILL-CAR.
RECYCLE 292:[C[?3][‘279][‘291]]
             ASSIGNMENT: 293.
RECYCLE 279:[A[?4][?1000][?0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 293.
```

```
RECYCLE 279:[A[?3][?1000][?0]]
 POP:
          PLACE: 80, EXP: 0, ENV: 0, REVAL: 0, MODE: 37, STACK: 286
RESTORE
            ASSIGNMENT: 0.

 POP:
          PLACE: 7, EXP: 292, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
-=>CAR
            ASSIGNMENT: 279.
            ASSIGNMENT: 279.
            ASSIGNMENT: 0.
RECYCLE 279:[A[?4][?1000][?0]]
-=>1000RECYCLE 279:[A[?3][?1000][?0]]


EXIT.
-=>-=>-=> ENTER READ LOOP.  Q IS 243
 EXIT
            ASSIGNMENT: 0.
RECYCLE 243:EXIT
            ASSIGNMENT: 0.
RECYCLE 276:[C[?3]['275]['0]]

-=>-=>-=>-=> LEAVING.
NODES DISPOSED, 18
NODES RECYCLED, 3
AVAIL--> 286



15.32.09.  SDJZ3VN   END OF LISTING
```

```
RECYCLE 280:[C[?5][*279][*282]]
 POP:
          PLACE: 9, EXP: 280, ENV: 0, REVAL: 0, MODE: 37, STACK: 287
-=>CDR
            ASSIGNMENT: 282.

RECYCLE 280:[C[?4][*279][*282]]
 POP:
          PLACE: 72, EXP: 280, ENV: 0, REVAL: 282, MODE: 37, STACK: 286
-=>EVLIS1
           DOTPAIR--> 287--> [[][163][280]].
           DOTPAIR--> 274--> [[][151][282]].
           CONS[292]-->[C[?0][*289][*291]]
            ASSIGNMENT: 292.
RECYCLE 282:[C[?4][*281][*284]]

RECYCLE 280:[C[?4][*279][*282]]
 POP:
          PLACE: 25, EXP: 169, ENV: 0, REVAL: 292, MODE: 37, STACK: 0
-=>ANB
            ASSIGNMENT: 169.
            ASSIGNMENT: 292.
      APPLY 169 TO 292
     PUSH-1: 286--> [S[*292][?7][*0]]

RECYCLE 169:FIRST
 POP:
          PLACE: 7, EXP: 292, ENV: 0, REVAL: 292, MODE: 37, STACK: 0
-=>CAR
     PUSH-1: 286--> [S[*292][?7][*0]]
     PUSH-1: 293--> [S[*0][?80][*286]]

 CONTEXT PUSH: 294 --> [*293][*292][*0]], PROCESS-CAR.

RECYCLE 292:[C[?5][?18][*291]]
 POP:
          PLACE: 80, EXP: 0, ENV: 0, REVAL: 292, MODE: 19, STACK: 288
-=>RESTORE
            ASSIGNMENT: 0.

 POP:
          PLACE: 1, EXP: 287, ENV: 0, REVAL: 292, MODE: 19, STACK: 0
-=>TOP
     PUSH-1: 288--> [S[*279][?1][*0]]

RECYCLE 287:[C[?1][*163][*280]]
--> AVAIL
 POP:
          PLACE: 1, EXP: 279, ENV: 0, REVAL: 292, MODE: 19, STACK: 0
-=>TOP
            ASSIGNMENT: 279.
RECYCLE 292:[C[?4][?18][*291]]

CONTEXT POP: [*293][*292][*0]]; FILL-CAR.
RECYCLE 292:[C[?3][*279][*291]]
            ASSIGNMENT: 293.
RECYCLE 279:[A[?4][?1000][?0]]

-=>-=>-=> EVALUATION. PROCESS STACK IS NODE 293.
```

```
RECYCLE 279:[A[?3][?1000][?0]]
 POP:
          PLACE: 80, EXP: 0, ENV: 0, REVAL: 0, MODE: 37, STACK: 286
RESTORE
            ASSIGNMENT: 0.

 POP:
          PLACE: 7, EXP: 292, ENV: 0, REVAL: 0, MODE: 37, STACK: 0
-=>CAR
            ASSIGNMENT: 279.
            ASSIGNMENT: 279.
            ASSIGNMENT: 0.
RECYCLE 279:[A[?4][?1000][?0]]
-=>1000RECYCLE 279:[A[?3][?1000][?0]]


EXIT.
-=>-=>-=> ENTER READ LOOP.  Q IS 243
 EXIT
            ASSIGNMENT: 0.
RECYCLE 243:EXIT
            ASSIGNMENT: 0.
RECYCLE 276:[C[?3]['275]['0]]

-=>-=>-=>-=> LEAVING.
NODES DISPOSED, 18
NODES RECYCLED, 3
AVAIL--> 286



15.32.09.  SDJZ3VN   END OF LISTING
```

```
      PASCAL COMPILER - E.T.H. ZURICH, SWITZERLAND
  PASCAL 6600 - 2.2                77/08/22. 22.51.53.
    INDIANA UNIVERSITY  WRUBEL COMPUTING CENTER  (76/06/21)

000006 PROGRAM SLISP(INPUT,OUTPUT);
000464 (* THIS IS AN INTERPRETER FOR A PROGRAMMING LANGUAGE BASED
000464    ON SUSPENDED CONSTRUCTION.  GENERAL COMMENTARY IS FOUND
000464    IN "AN INTERPRETIVE MODEL FOR A LANGUAGE BASED ON SUSPENDED
000464    COMPUTATION"; M.S. THESIS BY STEVEN D. JOHNSON, INDIANA
000464    UNIVERSITY, AUGUST, 1977.  CODE COMMENTS SUCH AS 'SECTION 3,4',
000464    REFER TO DISCUSSIONS IN THAT PAPER *)
000464 LABEL   1,2,3,4;
000464 CONST   MAXPTR = 131071; (*THIS IS (2**17)-1 FOR 17-BIT FIELD*)
000464         MAXNUM = 65535;  (*THIS IS (2**16)-1 TO PERMIT NEG. NOS.*)
000464         INFINITY = 65535;  (* HAS TO DO WITH RESOURCE ALLOCATION *)
000464         MEMORYSIZE = 7000;
000464         OBLISTSIZE = 128;
000464         HASHCONST = 87;
000464         BLANK = " ";
000464         NAMELENGTH = 7; (* MAX THAT FITS IN ONE NODE*)
000464         INPUTSIZE = 72;
000464         OUTPUTSIZE = 72;
000464         NIL = 0;
000464         LPREN = 1; (*CODES FOR READ PUSHES *)
000464         LANGLE = 2;
000464         LBRAC = 3;
000464         DOT = 4;
000464         QUOT = 5;
000464         STA = 6; (*END READ CODES*)
000464         TOP = 1; (* CODES FOR EVAL PUSHES *)
000464         ISFOUND = 2; (* SOME OF THESE EVAL-PROCEDURES (SEE SECTION 3.4)
000464                        HAVE BEEN MODIFIED OUT OF THE PROGRAM. *)
000464         ASSOC = 3;
000464         ASSOC1 = 4;
000464         ASSOC2 = 5;
000464         ASSOC3 = 6;
000464         CAR = 7;
000464         KICKAR = 8;
000464         CDR = 9;
000464         KICKDR = 10;
000464         LOOK = 11;
000464         TSTAR = 12;
000464         EQ1 = 13;
000464         EQ2 = 14;
000464         EQ3 = 15;
000464         ATOM = 16;
000464         PRINT = 17;
000464         PUTCAR = 18;
000464         PUTCDR = 19;
000464         APPLY1=20;
000464         LASTLINE=21;
000464         STARRED = 22;
000464         COND = 23;
000464         COND1 = 24;
000464         ANB = 25;
000464         DE = 26;
000464         DE1 = 27;
000464         APPLY = 28;
```

```
000464          EPHEMERAL = 29;
000464          ETERNAL = 30;
000464          DC = 31;
000464          FNSTAR = 32;
000464          FNSTAR1 = 33;
000464          FNSTAR2 = 34;
000464          FNSTAR3 = 35;
000464          FNSTAR4 = 36;
000464          ALLSTAR = 37;
000464          ALLSTAR1 = 38;
000464          ANYNULL = 39;
000464          ANYNULL1 = 40;
000464          ANYNULL2 = 41;
000464          APPLY2 = 42;
000464          APPLY3 = 43;
000464          APPLY4 = 44;
000464          NTH = 45;
000464          NTH1 = 46;
000464          ACAR = 47;
000464          ACDR = 48;
000464          ASTAR = 49;
000464          ASTAR1 = 50;
000464          EQ = 51;
000464          CARLIS4 = 52;
000464          ADD1 = 56;
000464          PLUS = 57;
000464          PLUS1 = 58;
000464          PLUS2 = 59;
000464          LAMBDA = 60;
000464          CHECK1 = 61;
000464          CHECK2 = 62;
000464          TEST1 = 63;
000464          TEST2 = 64;
000464          CARLIS = 65;
000464          CARLIS1 = 66;
000464          CARLIS2 = 67;
000464          CARLIS3 = 68;
000464          CDRLIS = 69;
000464          CDRLIS1 = 70;
000464          EVLIS = 71;
000464          EVLIS1 = 72;
000464          PAIRLIS = 73;
000464          PAIRLIS1 = 74;
000464          CHECK0 = 75;
000464          ANYNULL0=77;
000464          SUB1=78;
000464          RESTORE = 80;
000464 TYPE     PTR = 0..MAXPTR;  (* SEE SECTION 6.4 FOR DISCUSSION *)
000464          NUM = -MAXNUM..MAXNUM;
000464          REFERENCE = PACKED RECORD
000464                  CASE NUMBERP : BOOLEAN OF
000464                   FALSE:(RR:PTR);
000464                   TRUE:(NN:NUM)
000464                  END;      (* REFERENCE *)
000464          NODE = PACKED RECORD      (* FITS IN A 60-BIT WORD *)
000464                  MULTI:BOOLEAN;    (* MARKS FERN ORDER BY CONVERGENCE *)
000464                  ATOMP:BOOLEAN;    (* REDUNDANT-CHECK CDR.PNAME *)
000464                  EXTRA:0..7;       (* NOT USED-SAVE FOR WAITE-SHORR *)
```

```
000464                 CASE PNAME : BOOLEAN OF
000464                      FALSE:(REF:REFERENCE;
000464                             CAR:REFERENCE;
000464                             CDR:REFERENCE);
000464                       TRUE:(LENGTH:0..7;
000464                             EXTRAA:0..7;
000464                                CHR:PACKED ARRAY[1..8] OF CHAR);
000464                 END; (* NODE *)
000464                 PUTCONTROL = (TERPRI,PRINC);
000464         QQQ = PACKED ARRAY[1..8] OF CHAR;
000464 VAR     MEMORY : ARRAY[0..MEMORYSIZE] OF NODE; (* SECTION 6.3 *)
016215         NAME : ARRAY[0..NAMELENGTH] OF CHAR;
016225         INIMAGE : ARRAY[1..INPUTSIZE] OF CHAR;
016335         OUTIMAGE : ARRAY[1..OUTPUTSIZE] OF CHAR;
016445    (* CELL TYPE TEMPLATES *)
016445         NEWSUSPEND,NEWNUM,NEWATOM,NEWPNAME,NEWCONS,
016445           READPUSH,STACKPUSH,PRINTPUSH,RECOVERPUSH,
016445           SCRATCHNODE,PNODE : NODE;
016460    (* VALUE- AND INSPECTION-REGISTERS; SECTIONS 3.3,3.4,6.3,6.4 *)
016460         HEAPBOT,STACKTOP,AVAIL,REVAL,ASSOCVAR,
016460           ASSOCLIST,FULFILL,EXP,ENVDOT,ENVIRON,ALIST,FP,AP,
016460           DMY,DMY1,POINT,POINT1,OLDPOINT,PT1,PT2,PT3,
016460           FN,ARGS,MEXP,MENVDOT,FLIST,PRS,
016460           VARB,VAL,TEM,CARFN,CDRFN,LST,
016460           LIS,AEXP,STACK,P,Q : PTR;
016526    (* GLOBAL INTEGERS *)
016526         I,J,SPEAK,NLENGTH,INPOINT,OUTPOINT,N,NBR,
016526         CHARCOUNT,CALLRECLAIM,ANSWER,RETURNS,DRETURNS,TRACE,
016526         MEMORYLIMIT: INTEGER;
016545    (* GLOBAL PREDICATES *)
016545         LIST,FINIS,DONE,READACHAR,CARRAIGERETURN,DORECLAIM:BOOLEAN;
016553         THISCH,VCH:CHAR;
016555    (* SYSTEM ATOMS.  INITIALIZED IN THE MAIN BODY *)
016555         QNIL,QENV, QTRACE,QT,QFALSE,QCLIST,QMLIST,QASSOC,FAPPLY,
016555           QEVCAR,QEQ,QATOM,QCAR,QCDR,QCONS,QFONS,QCOND,QSTAR,QEVLIS,
016555           QPAIRLIS,QCARLIS,QCDRLIS,QOBLIST,QAND, QOR, QNULL,
016555           FCAR,FCADR,FCDR,FCDAR,
016555           QDE, QDC, QETERNAL, QEPHEMERAL, QNOT, QPLUS, QTIMES,
016555           QDIV,QMOD,QSUB1,QDIFF,QADD1,QUNDEFINED,QLPREN,JAWS,
016555           QEXP,QHSH,QSPEAK,QSTOP,QARGS,QFUNCTIO,QQUOTE,MACROQUOTE,
016555           MACRODOT,QFUNARG,QLAMBDA,QREDEF,QSQB,QCOLON,QUNBOUND,
016555           QIF,QTHEN,QELSE,QELSEIF,
016555           QAP,QFP,QSTARRED,QLABEL,QLIS,QLESS,QGREAT: PTR;
016664
016664
016664
016664
016664 PROCEDURE PUTAT(P:PTR); FORWARD;
000004
000004
000004
000004 PROCEDURE WRITENODE(P:PTR; ENDLN:BOOLEAN);
000005                                 (* DEBUG AID.  OUTPUTS A NODE IN
000005                                    THE FORMAT [[ REF ][ CAR ][ CDR ]] *)
000005     BEGIN
000005     IF MEMORY[P].ATOMP AND (NOT MEMORY[P].CAR.NUMBERP) THEN PUTAT(P)
000021     ELSE
000023         BEGIN
```

```
000023          IF MEMORY[P].ATOMP THEN WRITE("[A")
000034          ELSE IF MEMORY[P].MULTI THEN WRITE("[M")
000046          ELSE IF MEMORY[P].REF.NUMBERP THEN WRITE("[C")
000060          ELSE IF MEMORY[P].CAR.NUMBERP THEN WRITE("[S");
000072          IF MEMORY[P].REF.NUMBERP THEN WRITE("[#",MEMORY[P].REF.NN:1)
000113          ELSE WRITE("[^",MEMORY[P].REF.RR:1);
000130          IF MEMORY[P].CAR.NUMBERP THEN WRITE("][#",MEMORY[P].CAR.NN:1)
000151          ELSE WRITE("][^",MEMORY[P].CAR.RR:1);
000166          IF MEMORY[P].CDR.NUMBERP THEN WRITE("][#",MEMORY[P].CDR.NN:1)
000207          ELSE WRITE("][^",MEMORY[P].CDR.RR:1);
000224          WRITE("]]")
000231          END;
000231      IF ENDLN THEN WRITELN
000232      END;
000253
000253
000253
000253 (* THE NEXT SIX ROUTINES COMPRISE THE MEMORY MANAGEMENT KERNAL
000253    DISCUSSED IN SECTIONS 3.3, 6.3, AND 6.4.  SEE ALSO THE
000253    PROCEDURE 'SETREG'. *)
000253
000253 PROCEDURE DISPOZE(N:PTR);    (* RETURN A NODE TO AVAIL*)
000004      VAR VISIT:NODE;
000005      BEGIN
000005      IF (N=NIL) THEN BEGIN END
000010      ELSE IF MEMORY[N].ATOMP AND (NOT MEMORY[N].CAR.NUMBERP) THEN
000020          BEGIN
000020          (* INTERN RECOVERS LITERALS *)
000020          DRETURNS:= DRETURNS+1;
000022          MEMORY[N].REF.NN:= 0
000023          END
000026      (* THERE MAY BE OTHER 'SUB-ATOMIC' STRUCTURES, E.G.
000026         SOME STACKS MAY BE RECOVERABLE AS A WHOLE.  THIS
000026         IS TAKEN CARE OF AT THIS POINT. *)
000026      ELSE
000027          BEGIN
000027          VISIT:= NEWCONS;
000030          VISIT.CDR.RR:= AVAIL;
000034          MEMORY[N]:= VISIT;
000037          AVAIL:= N;
000041          DRETURNS:= DRETURNS+1
000041          END
000042     END;
000047
000047
000047
000047 PROCEDURE RECYCLE ( P:PTR );        (* FOLLOWS CDRS UNTIL A HIGH
000004                                      REFERENCE COUNT.  THE RESULT
000004                                      GOES TO AVAIL.  NEWNODE
000004                                      RECYCLES THE OTHER FIELDS *)
000004      LABEL 1,2;
000004      VAR CURSOR,TRAILER:PTR; CNODE:NODE;
000007      BEGIN
000007      IF (P=NIL) THEN GOTO 2;
000010      IF (TRACE>10) THEN WRITE("RECYCLE ",P:1,":");
000025      CURSOR:= P; CNODE:= MEMORY[P];
000034      IF CNODE.REF.NUMBERP AND (CNODE.REF.NN>1) THEN
000037          BEGIN
```

```
000037          IF (TRACE>10) THEN WRITENODE(CURSOR,TRUE);
000046          CNODE.REF.NN:= CNODE.REF.NN-1;
000055          MEMORY[P]:= CNODE
000057          END
000060      ELSE IF CNODE.ATOMP THEN DISPOZE(P)
000065      ELSE
000067          BEGIN
000067          WHILE TRUE DO
000070              BEGIN
000070              RETURNS:= RETURNS+1;
000071              IF TRACE>10 THEN WRITENODE(CURSOR,TRUE);
000101              TRAILER:= CURSOR;
000104              CURSOR:= CNODE.CDR.RR; CNODE:= MEMORY[CURSOR];
000113              IF (CURSOR=NIL) OR (CURSOR=TRAILER) THEN GOTO 1;
000116              IF CNODE.REF.NUMBERP AND (CNODE.REF.NN>1) THEN
000122                  BEGIN
000122                  CNODE.REF.NN:= CNODE.REF.NN-1;
000130                  MEMORY[CURSOR]:= CNODE;
000133                  GOTO 1
000134                  END;
000134               IF CNODE.ATOMP THEN
000136                  BEGIN
000136                  IF (TRACE>10) THEN
000140                      BEGIN
000140                      WRITENODE(CURSOR,FALSE);
000146                      WRITELN("<ATOM>")
000153                      END;
000154                  DISPOZE(CURSOR);
000160                  GOTO 1
000161                  END
000161              END;   (* WHILE LOOP *)
000162      1:  MEMORY[TRAILER].CDR.RR:= AVAIL;
000172          IF TRACE>10 THEN WRITELN("--> AVAIL");
000202          AVAIL:= P
000202          END;
000206      2:  END;
000222
000222
000222
000222 PROCEDURE NUDGE(P:PTR);    (* UPS THE REFERENCE COUNT OF FERN CELLS *)
000004          BEGIN
000004          IF (NOT (P=NIL)) AND MEMORY[P].REF.NUMBERP THEN
000014             MEMORY[P].REF.NN:= MEMORY[P].REF.NN+1
000017          END;
000034
000034
000034
000034 FUNCTION SUSPENDED(X:PTR):BOOLEAN;
000004      BEGIN
000004      SUSPENDED:= NOT MEMORY[X].REF.NUMBERP
000011      END;
000022
000022
000022
000022 FUNCTION ISATOM(P:PTR):BOOLEAN;
000004      BEGIN
000004      ISATOM:= (P=NIL) OR MEMORY[P].ATOMP
000011      END;
```

```
000024
000024
000024
000024 FUNCTION NEWNODE:PTR;       (* RETURN THE FIRST NODE ON THE AVAIL
000003                                 LIST.  IF AVAIL IS EMPTY WE'RE
000003                                 OUT OF MEMORY.  RECYCLE THE GARBAGE
000003                                 IN THE CAR AND REF FIELDS *)
000003     VAR RESULT:PTR; VISIT:NODE;
000005     BEGIN
000005     IF NOT (AVAIL=NIL) THEN
000007          BEGIN
000007          RESULT:= AVAIL; VISIT:= MEMORY[RESULT];
000016          AVAIL:= VISIT.CDR.RR;
000020          IF NOT VISIT.REF.NUMBERP THEN RECYCLE(VISIT.REF.RR);
000025          IF NOT VISIT.CAR.NUMBERP THEN RECYCLE(VISIT.CAR.RR)
000033          END
000034     ELSE
000035          BEGIN
000035          WRITELN; WRITELN;
000037          WRITELN(" -=>-=>-=> MEMORY IS EXHAUSTED. ");
000045          WRITELN("     -=> YOU HAVE SPECIFIED ",MEMORYLIMIT:1," NODES, ");
000064          WRITE("AND THE LIMIT IS ",MEMORYSIZE:1,".");
000077          WRITELN;WRITELN;WRITELN;HALT
000102          END;
000103     MEMORY[RESULT]:= NEWCONS;
000107     NEWNODE:= RESULT
000107     END;
000131
000131
000131
000131 PROCEDURE WARNING;    (* THIS PROCEDURE PRINTS AN WARNING
000003                          MESSAGE ABOUT EXCESSIVELY LARGE ATOMS. *)
000003     BEGIN
000003     WRITELN("<==-");
000014     WRITELN("LITERAL OR NUMERIC EXCEEDS BOUNDS.")
000021     END;
000032
000032
000032
000032 (* THE FOLLOWING ROUTINES FOR ATOM INTERNALIZATION AND
000032    INPUT-OUTPUT ARE ESSENTIALLY WRITTEN BY BROWN IN 1976,
000032    WITH MINOR MODIFICATIONS FOR RECENT REPRESENTATIONAL
000032    CHANGES *)
000032
000032 PROCEDURE PUTCH(MODE:PUTCONTROL;SYMBOL:CHAR);
000005 (*THIS FUNCTION MAINTAINS AN OUTPUT BUFFER WHICH IS WRITTEN
000005   OUT WHEN FULL OR WHEN THE FUNCTION IS CALLED WITH THE
000005   MODE SET TO TERPRI.*)
000005 VAR      I:INTEGER;
000006 BEGIN
000006          IF MODE=TERPRI THEN OUTPOINT:=OUTPUTSIZE;
000011          IF OUTPOINT=OUTPUTSIZE THEN
000013          BEGIN
000013           WRITELN;
000014           WRITE(BLANK);
000016           OUTPOINT:=1;(*ALLOWS FOR ONE LEADING BLANK*)
000017          END;
000017          IF MODE=PRINC THEN
```

```
000021          BEGIN
000021           OUTPOINT :=OUTPOINT+1;
000022           WRITE(SYMBOL);
000026          END;
000026 END;
000037
000037
000037
000037 PROCEDURE PUTPNAME(POINT1:PTR);
000004 (*THIS PROCEDURE PASSES THE CHARACTERS OF THE PNAME OF THE ATOM
000004   POINTED TO BY POINT1 TO PUTCH*)
000004 VAR      J : INTEGER;
000005          POINT : PTR;
000006          NAMEBUF : ARRAY[0..NAMELENGTH] OF CHAR;
000016          TEMP : PACKED ARRAY[1..8] OF CHAR;
000017 BEGIN
000017          POINT :=MEMORY[POINT1].CDR.RR;
000015          IF NOT MEMORY[POINT].PNAME THEN
000021          BEGIN
000021           WARNING;
000022           PUTCH(PRINC,BLANK);
000030          END
000030          ELSE BEGIN
000031           TEMP:=MEMORY[POINT].CHR;
000035           UNPACK(TEMP,NAMEBUF,0);
000040           J:=MEMORY[POINT].LENGTH;
000044           IF (OUTPUTSIZE-OUTPOINT)<J THEN PUTCH(TERPRI,BLANK);
000055           FOR I:=0 TO J DO
000057                 PUTCH(PRINC,NAMEBUF[I]);
000075          END;
000075 END;
000106
000106
000106
000106 PROCEDURE PUTNUM(N:INTEGER);
000004 (*THIS PROCEDURE CALCULATES THE CHARACTERS FOR A NUMBER TO
000004   BE OUTPUT AND SENDS THEM TO THE PUTCH ROUTINE*)
000004 VAR      I,J:INTEGER;
000006          NUMBUF:ARRAY[1..20] OF CHAR;
000032 BEGIN
000032          IF N<0 THEN
000010          BEGIN
000010           N:=-N;
000011           PUTCH(PRINC,"-");
000016          END;
000016          IF N=0 THEN PUTCH(PRINC,"0")
000025          ELSE BEGIN
000027          J:=0;
000030           WHILE N>0 DO
000032           BEGIN
000032            I:=N MOD 10;
000037            N:=N DIV 10;
000045            NUMBUF[20-J]:= CHR(I+ORD("0"));
000054            J:=J+1;
000055           END;
000055           FOR I:=20-J+1 TO 20 DO
000057                 PUTCH(PRINC,NUMBUF[I]);
000074          END;
```

```
000074 END;
000105
000105
000105
000105 FUNCTION GETCH:CHAR;     (* FETCH A CHARACTER FROM INPUT *)
000003    VAR VALUE:CHAR;
000004    BEGIN
000004    IF CARRAIGERETURN THEN    (* NOW AT END-OF-LINE *)
000007       BEGIN
000007       IF TRACE>0 THEN WRITELN;
000011       READLN;
000012       CARRAIGERETURN:= FALSE;
000013       READ(VALUE)
000020       END
000020    ELSE IF EOLN THEN    (* AT END OF LINE, SET FLAG AND RETURN A BLANK *
000022       BEGIN
000022       CARRAIGERETURN:= TRUE;
000023       VALUE:= BLANK
000023       END
000024    ELSE           (* NORMALLY JUST GET A CHARACTER *)
000024       READ(VALUE);
000031    IF (TRACE>0) THEN WRITE(VALUE);
000037    GETCH:= VALUE
000037    END;
000047
000047
000047
000047 FUNCTION READNUM:NUM;
000003 (*READNUM EXPECTS THE FIRST CHARACTER OF THE NUMBER IN THISCH.
000003   IT RETURNS THE NUMERIC VALUE OF THE NUMBER *)
000003 VAR      EXTRA:NUM;
000004          BEGIN
000004          I:=0;
000007          EXTRA :=ORD("0");
000013          WHILE THISCH IN ["0".."9"] DO
000016          BEGIN
000016           I := I*10+ORD(THISCH)-EXTRA;
000021           THISCH :=GETCH;
000025          END;
000025          IF I>MAXNUM THEN BEGIN
000027           WARNING;
000030           I := MAXNUM;
000031          END;
000031          READNUM:=I;
000036          END;
000044
000044
000044
000044 FUNCTION MAKENUM(X:NUM):PTR;
000004 (*MAKENUM CREATES A NUMBER NODE CONTAINING ITS PARAMETER AND
000004   RETURNS A POINTER TO IT *)
000004 VAR      POINT : PTR;
000005 BEGIN
000005          IF (X>MAXNUM) THEN BEGIN
000010            WARNING;
000011            X:=MAXNUM;
000012          END;
000012          IF (X<-MAXNUM)THEN
```

```
000015          BEGIN
000015           WARNING;
000016           X:=-MAXNUM
000016          END;
000023          POINT :=NEWNODE;
000027          SCRATCHNODE := NEWNUM;
000030          SCRATCHNODE.CAR.NN := X;
000035          MEMORY[POINT] := SCRATCHNODE;
000040          MAKENUM:=POINT;
000043 END;
000052
000052
000052
000052 FUNCTION HASH:PTR;
000003 (*HASH ASSUMES THE CHARACTERS OF THE ATOM ARE IN NAME AND THE
000003   LENGTH -1 IS IN NLENGTH.  IT RETURNS A POINTER TO
000003   THE APPROPRIATE OBLIST BUCKET.*)
000003 VAR      I,J:INTEGER;
000005 BEGIN
000005          J:=0;
000007          FOR I:=0 TO NLENGTH DO
000011          BEGIN
000013           J := J+ORD(NAME[I]);
000016           J := J* HASHCONST;
000020           J := J MOD OBLISTSIZE;
000021          END;
000022          HASH := J+1;
000026 END;
000035
000035
000035
000035 FUNCTION INTERN:PTR;
000003 (* INTERN ASSUMES THE CHARACTERS OF THE ATOM ARE IN THE ARRAY
000003    NAME AND THE LENGTH -1 IS IN NLENGTH.  INTERN CREATES A NEW
000003    ATOM NODE IF NECESSARY, AND RETURNS A POINTER TO THE ATOM*)
000003 VAR      PPOINT, OLDPOINT,NEWPOINT:PTR;
000006          SQUASH: PACKED ARRAY[1..8] OF CHAR;
000007          FOUND:BOOLEAN;
000010 BEGIN
000010          POINT:=HASH;(*THE INDEX OF THE PROPER OBLIST BUCKET*)
000012          PACK(NAME,0,SQUASH);(*SEARCH FOR THE ATOM*)
000017          PPOINT:= NIL;
000020          FOUND :=FALSE;
000021          WHILE (NOT (POINT=0)) AND (NOT FOUND) DO
000025          BEGIN
000025           OLDPOINT :=POINT;
000030           POINT := MEMORY[POINT].CAR.RR;
000036           IF NOT(POINT=0) THEN
000036           BEGIN
000036             IF MEMORY[MEMORY[POINT].CDR.RR].CHR = SQUASH THEN FOUND:= TRUE
000047             ELSE IF MEMORY[POINT].REF.NN = 0 THEN PPOINT:= OLDPOINT;
000057            END;
000057           END;
000060          IF POINT = 0 THEN
000061             BEGIN
000061             IF PPOINT = NIL THEN
000063                BEGIN
000063                POINT:= NEWNODE;
```

```
000067                PPOINT:= NEWNODE;
000073                MEMORY[OLDPOINT].CAR.RR:= POINT
000075                END
000103             ELSE
000103                BEGIN
000103                POINT:= MEMORY[PPOINT].CAR.RR;
000111                PPOINT:= MEMORY[POINT].CDR.RR
000113                END;
000116             SCRATCHNODE:= NEWATOM;
000120             SCRATCHNODE.CDR.RR:= PPOINT;
000124             MEMORY[POINT]:= SCRATCHNODE;
000127             SCRATCHNODE:= NEWPNAME;
000131             SCRATCHNODE.LENGTH:= NLENGTH;
000135             SCRATCHNODE.CHR:= SQUASH;
000137             MEMORY[PPOINT]:= SCRATCHNODE;
000142             END;
000142          INTERN:=POINT;
000146 END;
000161
000161
000161
000161 PROCEDURE PUTAT (* P:PTR *);   (* SELECTS PUTPNAME OR PUTNUM *)
000004     BEGIN
000004     IF P=NIL THEN PUTPNAME(QNIL)
000013     ELSE IF MEMORY[P].ATOMP AND (NOT MEMORY[P].CAR.NUMBERP)
000023         THEN PUTPNAME(P)
000027     ELSE PUTNUM(MEMORY[P].CAR.NN)
000034     END;
000042
000042
000042
000042
000042
000042
000042 FUNCTION MYREAD:PTR;     (* PARSE INPUT INTO AN S-EXPRESSION *)
000003 TYPE    TTYPE = (TLETTER,TDIGIT,TWEIRD,TRPREN,TLPREN,TCOLON,
000003                 TRANGLE,TLANGLE,TRBRAC,TLBRAC,TSIGN,TSTAR,
000003                 TQUOTE,TDOT,THSH,TCOMMENT,TCANCEL,TSLASH,
000003                 TCOMMA);
000003 VAR     CHTYPE : TTYPE;
000004          RSTACK,RESULT,TEMP: PTR;   (* RSTACK AND RESULT ARE
000007                                        GLOBAL TO THE READ PROCEDURES. *)
000007          RESTARTING, DOTTING: BOOLEAN;
000011         RNODE:NODE;
000012    (*  A LINKED STACK (BY REFERENCE-COUNT FIELD) OF SUBLISTS IS
000012        BUILT. CLOSING PARENTHESES CAUSE THE TOP LIST TO BE INCLUDED
000012        IN THE NEXT.  THE RESULT IS THE INTERNAL FORM.   *)
000012     PROCEDURE RPUSH(VAL:PTR);         (*PUSH A NEW SUBLIST
000004                                        ONTO THE READ STACK
000004                                        LINKED IN REF *)
000004         BEGIN
000004         RNODE:= READPUSH;
000011         RNODE.REF.RR:= RSTACK;
000015         RNODE.CAR.RR:= VAL;
000021         NUDGE(VAL);
000023         RSTACK:= NEWNODE;
000027         MEMORY[RSTACK]:= RNODE;
000033         END;
```

```
000040
000040
000040    PROCEDURE RPOP;            (* POP THE READ STACK *)
000003        (* GLOVAR RESULT . PUT THE TOP OF THE READ STACK INTO
000003           RESULT, AND POP THE STACK.  *)
000003        BEGIN
000003        RESULT:= RSTACK;
000011        RNODE:= MEMORY[RSTACK];
000015        RSTACK:= RNODE.REF.RR;  (* FIRST THE POP *)
000020        RNODE.REF.NUMBERP:= TRUE;  (* CHANGE REF-CNT TO A NUMBER *)
000022        RNODE.REF.RR:= 0;
000024        MEMORY[RESULT]:= RNODE
000026        END;
000031
000031
000031    PROCEDURE RESTART;       (* ON ERROR OR CANCELLATION BY THE USER
000003                                RETURN THE STACK TO AVAILABLE SPACE
000003                                AND NOTIFY THE MAIN LOOP *)
000003        BEGIN
000003        RECYCLE(RSTACK);
000012        RSTACK:= NIL;
000014        WHILE NOT CARRAIGERETURN DO THISCH:= GETCH;
000022        RESTARTING:= TRUE
000022        END;
000027
000027
000027    PROCEDURE RERROR(ERRTYPE:INTEGER); (* OUTPUT MESSAGE AND RESTART *)
000004        BEGIN
000004        WRITELN;
000010        WRITE("-=>-=>SYNTAX ERROR: ");
000015        CASE ERRTYPE OF
000022            1: WRITELN("INAPPROPRIATE DOT.");
000031            2: WRITELN("UNBALANCED SQUARE-BRACKET.");
000040            3: WRITELN("UNBALANCED ANGLE-BRACKET.");
000047            4: WRITELN("MISUSED STAR.");
000056            5: WRITELN("MISPLACED APPLICATION (':')");
000065            END;
000072        WHILE NOT (THISCH=".") DO THISCH:= GETCH;
000100        CARRAIGERETURN:= TRUE;
000101        RESTART;
000103        END;
000127
000127
000127
000127    PROCEDURE RLOOK;  (* INPUT LOOKAHEAD.  GET THE NEXT NON-BLANK
000003                         CHARACTER IN THISCH.  NOTIFY THE READ
000003                         LOOP THAT THIS HAS BEEN DONE.  *)
000003        BEGIN
000003        REPEAT THISCH:= GETCH UNTIL NOT ((THISCH=BLANK)OR(THISCH=","));
000016        READACHAR:= FALSE
000016        END;
000022
000022
000022
000022    PROCEDURE RBUILD;  (*TAKE THE CURRENT RESULT AND ADD IT TO THE
000003                          END OF THE TOP-MOST STACK ELEMENT.
000003                        THIS IS THE READ ROCESSOR'SPSEUDO
000003                        CONSTRUCTOR.  SEE SECTION 6.5, AND
```

```
000003                        THE DISCUSSION OF INPUT-OUTPUT IN CHAPTER
000003                        SEVEN *)
000003        LABEL 1;
000003        VAR TEMP1,TEMP2: PTR;
000005        BEGIN
000005        IF RSTACK = NIL THEN BEGIN END
000010        ELSE
000010            BEGIN
000010            WHILE MEMORY[RSTACK].CAR.RR = MACROQUOTE DO
000016                BEGIN
000016                MEMORY[RSTACK].CAR.RR:= QQUOTE;
000026                NUDGE(QQUOTE);              (* CREATE A CALL TO QUOTE *
000030                RECYCLE(MACROQUOTE);
000034                RNODE:= NEWCONS;
000036                RNODE.CAR.RR:= RESULT;
000043                RNODE.REF.NN:= 1;
000045                NUDGE(RESULT);
000047                TEMP1:= NEWNODE;
000053                MEMORY[RSTACK].CDR.RR:= TEMP1;
000062                MEMORY[TEMP1]:= RNODE;
000065                RPOP;     (* RESULT GETS THE TOP NODE IN THE STACK *)
000066                IF RSTACK=NIL THEN GOTO 1
000071                END;
000072            IF RSTACK=NIL THEN GOTO 1;
000074            TEMP1:= RSTACK;
000100            WHILE NOT (MEMORY[TEMP1].CDR.RR = NIL) DO
000105                TEMP1:= MEMORY[TEMP1].CDR.RR;
000113            IF MEMORY[TEMP1].CAR.RR = QLPREN THEN
000120                BEGIN
000120                RECYCLE(QLPREN);
000123                MEMORY[TEMP1].CAR.RR:= RESULT; NUDGE(RESULT)
000136                END
000137            ELSE
000140                BEGIN
000140                RNODE:= NEWCONS;
000142                RNODE.CAR.RR:= RESULT; NUDGE(RESULT);
000152                RNODE.REF.NN:= 1;
000156                RNODE.MULTI:= (MEMORY[RSTACK].CAR.RR=QMLIST);
000167                RESULT:= NEWNODE;  (* IN CASE OF SLASH *)
000173                MEMORY[TEMP1].CDR.RR:= RESULT;
000202                MEMORY[RESULT]:= RNODE;
000205                END;
000205            NUDGE(RESULT);
000212 1:         END
000212        END;
000221
000221
000221
000221    PROCEDURE RCOMP;      (* WE'VE ENCOUNTERED A LIST
000003                                 TERMINATOR ('>' OR ']').
000003                                 LOOK FOR A ':'.  IF FOUND
000003                                 PUSH ANOTHER CALL TO APPLY.
000003                                 OTHERWISE POP ALL CALLS TO APPLY.*)
000003        BEGIN
000003        RPOP;    (* THIS IS THE LIST STRUCTURE *)
000007        RLOOK;
000011        IF THISCH = ":" THEN
000013            BEGIN
```

```
000013              RPUSH(QCOLON);
000020              READACHAR:= TRUE;
000021              RBUILD
000021              END
000023          ELSE
000024              BEGIN
000024              RBUILD;
000026              WHILE MEMORY[RSTACK].CAR.RR=QCOLON DO
000034                  BEGIN
000034                  RPOP;
000035                  RBUILD
000035                  END
000037              END
000040          END;
000043
000043
000043
000043 BEGIN   (* THE PROCEDURE MYREAD SEE SECTION 6.5. *)
000043      RESTARTING:= TRUE;
000007      RESULT:= NIL;
000010      RSTACK:= NIL;
000010      WHILE RESTARTING OR (NOT (RSTACK = NIL)) DO
000014          BEGIN
000014          IF READACHAR OR ((THISCH=BLANK)OR(THISCH=",")) THEN RLOOK;
000023          READACHAR:= TRUE;
000024          RESTARTING:= FALSE;
000025          (* ASSIGN TYPES TO CHARACTERS*)
000025           CASE THISCH OF
000031            "A","B","C","D","E","F","G","H","I","J","K" ,
000031              "L","M","N","O","P","Q","R","S","T","U","V",
000031              "W","X","Y","Z": CHTYPE := TLETTER;
000033           "0","1","2","3","4","5","6","7","8","9":
000033                CHTYPE := TDIGIT;
000035            "(": CHTYPE := TLPREN;
000037            ")": CHTYPE := TRPREN;
000041            "<" :CHTYPE := TLANGLE;
000043            ">": CHTYPE := TRANGLE;
000045            "[": CHTYPE := TLBRAC;
000047            ":": CHTYPE:= TCOLON;
000051            "#": CHTYPE:= THSH;
000053            "]": CHTYPE := TRBRAC;
000055            ".": CHTYPE := TDOT;
000057            "+","-" : CHTYPE := TSIGN;
000061            "*": CHTYPE := TSTAR;
000063            "^": CHTYPE := TQUOTE;
000065            """": CHTYPE := TQUOTE;
000067            ";": CHTYPE:= TCOMMENT;
000071            "/": CHTYPE:= TSLASH;
000073"=","       ",","$","
        "      : CHTYPE := TWEIRD;
000075           END;
000174          (* THE READ LOOP MANIPULATES THE READ STACK ACCORDING
000174             TO THE CHARACTER MOST RECENTLY INPUT.  SURPRISINGLY
000174             A CASE STATMENT IS USED TO DETERMINE THE ACTION *)
000174          CASE CHTYPE OF
000201              TWEIRD,TLETTER: BEGIN   (*INTERN ATOMS, ADD TO SUBLIST*)
000201                    NLENGTH := -1;
000203                    FOR I := 0 TO NAMELENGTH DO NAME[I]:= BLANK;
000213                    I:=0;
```

```
000214            REPEAT
000214             IF I>NAMELENGTH THEN
000216             BEGIN
000216              WARNING;
000217              WHILE THISCH IN ["A".."9"] DO THISCH := GETCH;
000226            END
000226            ELSE BEGIN
000227             NAME[I] := THISCH;
000234             NLENGTH := NLENGTH +1;
000236             I := I+1;
000236             THISCH := GETCH;
000242             END;
000242           UNTIL NOT (THISCH IN ["A".."9"]);
000245          RESULT:=INTERN;
000251          READACHAR:= FALSE;
000252          IF RESULT=QNIL THEN RESULT:= NIL;
000254          IF (RESULT=QDE) AND (RSTACK=NIL) THEN RPUSH(QDE)
000263          ELSE IF (RESULT=QDC) AND (RSTACK=NIL) THEN RPUSH(QDC)
000273          ELSE
000275              BEGIN
000275              IF THISCH=BLANK THEN RLOOK;
000300              IF THISCH=":" THEN
000302                  BEGIN
000302                  RPUSH(QCOLON);
000306                  RBUILD;
000310                  READACHAR:= TRUE
000310                  END
000311              ELSE
000312                  BEGIN
000312                  RBUILD;
000314                  WHILE MEMORY[RSTACK].CAR.RR=QCOLON DO
000322                      BEGIN
000322                      RPOP;
000323                      RBUILD
000323                      END
000325                  END
000326              END
000326          END;
000327
000327  TSIGN, TDIGIT: BEGIN  (*NUMBER*)
000327          IF THISCH = "-" THEN
000331             BEGIN
000331             THISCH:= GETCH;
000335             RESULT:= MAKENUM(-READNUM)
000342             END
000346          ELSE IF THISCH = "+" THEN
000350             BEGIN
000350             THISCH:= GETCH;
000354             RESULT:= MAKENUM(READNUM)
000360             END
000364          ELSE
000364             RESULT:= MAKENUM(READNUM);
000374          READACHAR:= FALSE;
000375          IF THISCH=BLANK THEN RLOOK;
000400          IF THISCH=":" THEN
000402              BEGIN
000402              RPUSH(QCOLON);
000406              RBUILD;
```

```
000410                  READACHAR:= TRUE
000410                  END
000411              ELSE
000412                  BEGIN
000412                  RBUILD;
000414                  WHILE MEMORY[RSTACK].CAR.RR=QCOLON DO
000422                      BEGIN
000422                      RPOP;
000423                      RBUILD
000423                      END
000425                  END
000426              END;
000427      TCOLON:
000427          RERROR(5);
000432      THSH:
000432          BEGIN
000432          RESULT:= QHSH;
000436          RBUILD
000436          END;
000440      TLPREN:
000440          BEGIN
000440          RLOOK;
000442          IF THISCH=")" THEN
000444              BEGIN
000444              RESULT:= NIL;
000445              RBUILD
000445              END
000446          ELSE
000447              RPUSH(QLPREN)
000452          END;
000454      TLANGLE:
000454          RPUSH(QCLIST);
000461      TLBRAC:
000461          RPUSH(QMLIST);
000466      TSLASH:
000466          BEGIN
000466          RLOOK;
000470          IF THISCH = "/" THEN RESTART
000472          ELSE
000474              BEGIN
000474              RNODE:= MEMORY[RSTACK];
000500              IF (RNODE.CAR.RR=QMLIST) AND (NOT (RNODE.CDR.RR=NIL))
000503                  THEN MEMORY[RESULT].MULTI:= FALSE
000510              END
000514          END;
000515      TQUOTE:
000515          RPUSH(MACROQUOTE);
000522      TCOMMENT:
000522          BEGIN
000522          RESTARTING:= TRUE;
000523          WHILE NOT CARRAIGERETURN DO THISCH:= GETCH;
000531          END;
000532      TDOT:
000532          IF (RSTACK=NIL) THEN RESTARTING:= TRUE
000534          ELSE
000535              REPEAT
000535              RPOP;
000537              RBUILD
```

```
000537              UNTIL (RSTACK=NIL);
000544      TRPREN:
000544          BEGIN
000544          IF RSTACK = NIL THEN RESTARTING:= TRUE
000546          ELSE
000547              BEGIN
000547              RPOP;
000551              RBUILD;
000553              END
000553          END;
000554      TRBRAC:
000554          BEGIN
000554          IF RSTACK=NIL THEN RESTARTING:= TRUE
000556          ELSE IF MEMORY[RSTACK].CAR.RR = QMLIST THEN RCOMP
000564          ELSE
000566              BEGIN
000566              WHILE MEMORY[RSTACK].CAR.RR = QCOLON DO
000574                  BEGIN
000574                  RPOP;
000575                  RBUILD
000575                  END;
000600              IF MEMORY[RSTACK].CAR.RR = QMLIST THEN RCOMP
000605              ELSE RERROR(2)
000610              END
000611          END;
000612      TRANGLE:
000612          BEGIN
000612          IF RSTACK = NIL THEN RESTARTING:= TRUE
000614          ELSE IF MEMORY[RSTACK].CAR.RR = QCLIST THEN RCOMP
000622          ELSE
000624              BEGIN
000624              WHILE MEMORY[RSTACK].CAR.RR = QCOLON DO
000632                  BEGIN
000632                  RPOP;
000633                  RBUILD
000633                  END;
000636              IF MEMORY[RSTACK].CAR.RR = QCLIST THEN RCOMP
000643              ELSE RERROR(3)
000646              END
000647          END;
000650      TSTAR:
000650          IF RSTACK = NIL THEN RERROR(4)
000652          ELSE
000655              BEGIN
000655              RLOOK;
000657              IF (THISCH= ">") OR (THISCH= "]") THEN
000664                  BEGIN
000664                  RESULT:= RSTACK;
000667                  WHILE MEMORY[RESULT].CDR.RR<>NIL DO
000674                      RESULT:= MEMORY[RESULT].CDR.RR;
000702                  MEMORY[RESULT].CDR.RR:= RESULT;
000710                  END
000710              ELSE RERROR(4)
000712              END
000713      END;   (* OF THE CASE STATEMENT *)
000736      END;
000737                      (*END OF THE WHILE LOOP *)
000737      MYREAD:= RESULT
```

```
000737         END;                  (* END OF PROCEDURE MYREAD *)
000763
000763
000763
000763
000763
000763
000763 (* HERE FOLLOW SEVERAL AUXILIARY FIELD ACCESS FUNCTIONS FOR
000763    MANIFEST STRUCTURES.  SEE SECTION 7.2.  THESE, ALONG WITH
000763    CAR AND CDR AND REF SHOULD BE MADE MACINE SPECIFIC *)
000763
000763 FUNCTION CADR(EXP:PTR):PTR;
000004 (* THIS IS A :CADR*)
000004 VAR     TEMP:PTR;
000005 BEGIN
000005         TEMP:=MEMORY[EXP].CDR.RR;
000015         TEMP:=MEMORY[TEMP].CAR.RR;
000022         CADR:=TEMP;
000024 END;
000033 FUNCTION CAAR (EXP:PTR):PTR;
000004 VAR     TEMP:PTR;
000005 BEGIN
000005         TEMP:=MEMORY[EXP].CAR.RR;
000015         TEMP:=MEMORY[TEMP].CAR.RR;
000022         CAAR:=TEMP;
000024 END;
000033 FUNCTION CDDR(EXP:PTR):PTR;
000004 VAR     TEMP:PTR;
000005 BEGIN
000005         TEMP:=MEMORY[EXP].CDR.RR;
000015         CDDR:=MEMORY[TEMP].CDR.RR;
000022 END;
000031 FUNCTION CADAR (EXP:PTR):PTR;
000004 VAR     TEMP : PTR;
000005 BEGIN
000005         TEMP:=MEMORY[EXP].CAR.RR;
000015         TEMP:=MEMORY[TEMP].CDR.RR;
000022         TEMP:=MEMORY[TEMP].CAR.RR;
000027         CADAR:=TEMP;
000031 END;
000040 FUNCTION CADDR(EXP:PTR):PTR;
000004 VAR     TEMP:PTR;
000005 BEGIN
000005         TEMP:=MEMORY[EXP].CDR.RR;
000015         CADDR:=CADR(TEMP);
000024 END;
000033 FUNCTION CADDDR(EXP:PTR):PTR;
000004 VAR     TEMP:PTR;
000005 BEGIN
000005         TEMP:=MEMORY[EXP].CDR.RR;
000015         CADDDR:=CADDR(TEMP);
000024 END;
000033
000033
000033 PROCEDURE SETREG(VAR REG:PTR;VAL:PTR);  (* THIS IS A SYSTEM ASSIGNMENT
000005                                             STATEMENT WHICH INCLUDES
000005                                             THE ASSOCIATED ADJUSTMENTS
000005                                             IN REFERENCE COUNTS. *)
```

```
000005     BEGIN
000005     IF (TRACE>5) THEN WRITELN("            ASSIGNMENT: ",VAL:1,".");
000025     IF NOT (REG=VAL) THEN
000027         BEGIN
000027         NUDGE(VAL);
000033         RECYCLE(REG);
000040         REG:= VAL;
000044         END
000044     END;
000056
000056
000056
000056 FUNCTION PROCESS(FNAME:NUM; ARGUMENT:PTR):PTR;
000005                                         (** THIS WILL CREATE A
000005                                              STACK FOR EVALUA-
000005                                              TION PROCESSORS *)
000005     VAR STACK:PTR; PNODE: NODE;
000007     BEGIN
000007     PNODE:=NEWSUSPEND;
000011     STACK:= NEWNODE;
000015     PNODE.CAR.NUMBERP:= TRUE;
000017     PNODE.CAR.NN:= FNAME;
000025     PNODE.REF.RR:= ARGUMENT;
000031     NUDGE(ARGUMENT);
000033     MEMORY[STACK]:= PNODE;
000037     IF (TRACE>5) THEN
000041         BEGIN
000041         WRITELN;
000042         WRITE("-=>-=>-=>PROCESS CREATED: ");
000047         WRITENODE(STACK,TRUE)
000054         END;
000055     PROCESS:= STACK
000055     END; (*PROCEDURE PROCESS*)
000075
000075
000075
000075
000075 FUNCTION CONS(CAREXP,CDREXP,ENVIRONMENT:PTR):PTR;
000006                                        (* THIS IS THE SUSPENDING
000006                                           CONSTRUCTOR.   *)
000006     VAR PT:PTR; CNODE,CNODE1,CNODE2:NODE;
000012     BEGIN
000012     CNODE:= STACKPUSH; CNODE.CAR.NN:= TOP;
000013     CNODE1:= STACKPUSH; CNODE1.CAR.NN:= RESTORE;
000016     CNODE1.REF.RR:= ENVIRONMENT;
000022     CNODE2:= NEWCONS;          (* THESE ARE FUNCTION BODY CONSTANTS *)
000023     IF NOT(CAREXP=NIL) THEN
000024         (* BUILD AN EVALUATION STACK TO STICK IN THE CAR FIELD *)
000024         BEGIN
000024         NUDGE(ENVIRONMENT);
000027         NUDGE(CAREXP);
000033         CNODE.REF.RR:= CAREXP;
000040         PT:= NEWNODE; MEMORY[PT]:= CNODE;
000047         CNODE1.CDR.RR:= PT;
000052         PT:= NEWNODE; MEMORY[PT]:= CNODE1;
000061         CNODE2.CAR.RR:= PT
000061         END;
000065     IF NOT(CDREXP=NIL) THEN
```

```
000067          (* BUILD ANOTHER FOR THE CDR FIELD *)
000067          BEGIN
000067          NUDGE(ENVIRONMENT);
000073          NUDGE(CDREXP);
000077          CNODE.REF.RR:= CDREXP;
000104          PT:= NEWNODE; MEMORY[PT]:= CNODE;
000113          CNODE1.CDR.RR:= PT;
000116          PT:= NEWNODE; MEMORY[PT]:= CNODE1;
000125          CNODE2.CDR.RR:= PT
000125          END;
000130      PT:= NEWNODE; MEMORY[PT]:= CNODE2;
000137      IF (TRACE>12) THEN
000141          BEGIN
000141          WRITE("           ");
000146          WRITE("CONS[",PT:1,"]-->");
000164          WRITENODE(PT,TRUE);
000172          END;
000172      CONS:= PT
000172      END;
000215
000215
000215
000215 FUNCTION DOTPAIR(X,Y:PTR):PTR;
000005 (*OTHERWISE KNOWN AS :CONS*)
000005 VAR PT1,PT2:PTR;
000007     TNODE:NODE;
000010 BEGIN
000010          TNODE:=NEWCONS;
000011          TNODE.CAR.RR:=X;
000015          TNODE.CDR.RR:=Y;
000017          NUDGE(X);
000022          NUDGE(Y);
000026          PT1:=NEWNODE;
000032          MEMORY[PT1]:=TNODE;
000035          IF (TRACE>12) THEN
000037              BEGIN
000037              WRITE("           DOTPAIR--> ",PT1:1,"-->");
000055              WRITELN(" [[][",X:1,"][",Y:1,"]].")
000104              END;
000105          DOTPAIR:=PT1;
000111 END;
000133
000133
000133
000133 FUNCTION STAR(AEXP,ENVDOT:PTR):PTR;  (* A CONSTRUCTOR FOR
000005                                          STARRED STRUCTURES *)
000005 VAR TEM:PTR;
000006 BEGIN
000006          SPEAK:=SPEAK+1;
000011          TEM:=CONS(AEXP,NIL,ENVDOT);
000024          MEMORY[TEM].CDR.RR:=TEM;
000032          STAR:=TEM;
000034 END;
000045
000045
000045
000045 FUNCTION LISHELP(EXP:PTR):PTR;
000004  (*THIS FUNCTION CONSTRUCTS THE LIST (:CONS (LIST (:CONS
```

```
000004          ?EXP EXP)) ()) USED IN CARLIS AND CDRLIS*)
000004 VAR      POINT,PT1:PTR;
000006 BEGIN
000006          POINT:=DOTPAIR(QEXP,EXP);
000021          POINT:=DOTPAIR(POINT,NIL);
000032          POINT:=DOTPAIR(POINT,NIL);
000043          LISHELP:=POINT;
000045 END;
000056
000056
000056
000056 FUNCTION MAKEENV(ARGS:PTR):PTR;
000004 (* LIKE LISPHELP WITH ARGS INSTEAD OF EXP*)
000004 BEGIN
000004          POINT := DOTPAIR(QARGS,ARGS);
000021          POINT := DOTPAIR(POINT,NIL);
000032          MAKEENV:= DOTPAIR(POINT,NIL);
000043 END;
000050
000050
000050
000050 FUNCTION SYSATM(LETTERS:QQQ;LEN:INTEGER):PTR;
000005 (*THIS FUNCTION SETS UP THE SYSTEM ATOMS IN THE OBLIST.
000005   THEIR ADDRESSES ARE USED TO CONTROL PROGRAM FLOW.*)
000005 VAR      TNODE:PTR;
000006 BEGIN
000006          UNPACK(LETTERS,NAME,0);
000013          NLENGTH := LEN;
000014          TNODE:=INTERN;
000020          MEMORY[TNODE].REF.NN:=2;(*SO SYSTEM ATOMS
000025                                  ARE NEVER GARBAGE COLLECTED*)
000025          SYSATM:=TNODE;
000027 END;
000036
000036
000036
000036 FUNCTION SEARCH(LIST,TARGET:PTR):PTR;
000005                            (* THE FLIST AND ALIST ARE FULLY MANIFEST
000005                               STRUCTURES, SO SEARCHES AREN'T SUSPENSION-
000005                               SENSITIVE.  LOOK FOR A BINDING BY SEARCHING
000005                               TWO LISTS IN PARALLEL.  RETURN DOTTED
000005                               LIST CELLS, SO CALLING ROUTINE CAN REPLACE
000005                               VALUES WITHOUT ANOTHER SEARCH.  SET
000005                               'MULTI' FLAG TO INDICATE SUCCESS. *)
000005          VAR NAMES,VALUES:PTR;
000007          BEGIN
000007          NAMES:= MEMORY[LIST].CAR.RR;
000016          VALUES:= MEMORY[LIST].CDR.RR;
000022          WHILE (MEMORY[NAMES].CAR.RR <> TARGET) AND
000027               (MEMORY[VALUES].CDR.RR <> NIL)    DO
000035             BEGIN
000035             NAMES:= MEMORY[NAMES].CDR.RR;
000042             VALUES:= MEMORY[VALUES].CDR.RR
000044             END;
000050          VALUES:= DOTPAIR(NAMES,VALUES);
000062          MEMORY[VALUES].MULTI:= (MEMORY[NAMES].CAR.RR = TARGET);
000076          SEARCH:= VALUES
000076          END;
```

```
000114
000114
000114
000114
000114
000114
000114
000114
000114
000114 PROCEDURE EVAL(STACK:PTR;RESOURCES:INTEGER);
000005                (* EVALHELP TAKES A STACK AND PRESENTLY SUFFICIENT
000005                   RESOURCES TO FORCE FULL EVALUATION.  THIS
000005                   IS A PROCESSOR LOGICALLY.  NOTES (** --- **) (* LEAD
000005                   THE WAY TO LIMBO-CAR. *)
000005  VAR
000005     PT,PT1,PT2,STAQUE:PTR;
000011     PLACE,I,N,MODECODE:INTEGER; ALLDONE:BOOLEAN;
000016     (** LOCALIZE REGISTER NAMES:  REVAL, ENVDOT, ETC. **)
000016     ENODE: NODE;
000017
000017 PROCEDURE PUSHONE(NUM: INTEGER; PNT: PTR);
000005                (* THERE ARE TWO STACK-PUSH ROUTINES.  THE
000005                   FIRST PUSHES A PROCEDURE CALL (CAR) AND
000005                   A POINTER (CDR).  THE STACK IS LINKED BY
000005                   THE CDR FIELD. *)
000005        (* GLOVAR STACK (REFERENCE); GLOCON NEWSUSPEND (A TEMPLATE)
000005           GLOBAL PROCEDURE NEWNODE,NUDGE *)
000005        BEGIN
000005        ENODE:= NEWSUSPEND;
000011        ENODE.CAR.NUMBERP:= TRUE;
000013        ENODE.CAR.NN:= NUM;
000020        ENODE.REF.RR:= PNT;
000024        NUDGE(PNT);
000026        ENODE.CDR.RR:= STACK;
000033        STACK:= NEWNODE;
000037        MEMORY[STACK]:= ENODE;
000043        IF (TRACE>5) THEN
000045            BEGIN
000045             WRITE("     PUSH-1: ",STACK:1,"--> ");
000063            WRITENODE(STACK,TRUE)
000071            END;
000072        END;   (* END OF PROCEDURE PUSHONE *)
000104
000104
000104    PROCEDURE PUSHTWO(PNT1,PNT2: PTR);
000005                (* THIS PUSH PUTS TWO POINTERS INTO THE NODE *)
000005        (* GLOVAR STACK; GLOCON NEWSUSPEND; GLOBAL PROC NEWNODE,NUDGE *)
000005        BEGIN
000005        ENODE:= NEWSUSPEND;
000011        ENODE.REF.RR:= PNT1;
000015        ENODE.CAR.RR:= PNT2;
000021        NUDGE(PNT1);
000023        NUDGE(PNT2);
000027        ENODE.CDR.RR:= STACK;
000034        STACK:= NEWNODE;
000040        MEMORY[STACK]:= ENODE;
000044        IF (TRACE>5) THEN
000046            BEGIN
```

```
000046              WRITE("     PUSH-2: ",STACK:1,"--> ");
000064              WRITENODE(STACK,TRUE);
000073              END;
000073          END;         (* END OF PROCEDURE PUSHTWO *)
000105
000105
000105      PROCEDURE POP;            (* POPS A PUSH OF TYPE ONE *)
000003          (* GLOVAR PLACE: INTERGER; EXP, STACK: PTR *)
000003          BEGIN
000003          ENODE:= MEMORY[STACK];
000012          PLACE:= ENODE.CAR.NN;
000014          RECYCLE(EXP);           (* REUSE THE REFERENCE FROM THE STACK *)
000020          EXP:= ENODE.REF.RR;
000025          DISPOZE(STACK);
000031          STACK:= ENODE.CDR.RR;
000036          IF (TRACE>5) THEN
000040              BEGIN
000040              WRITELN(" POP:");
000046              WRITE("             PLACE: ",PLACE:1,", EXP: ",EXP:1);
000070              WRITE(", ENV: ",ENVDOT:1,", REVAL: ",REVAL:1);
000112              WRITELN(", MODE: ",MODECODE:1,", STACK: ",STACK:1)
000134              END
000135          END;    (* END OF PROCEDURE POP *)
000150
000150
000150  PROCEDURE LOAD(VAR REG1,REG2: PTR);
000005                             (* POPS A PUSH OF TYPE TWO.  ASSIGNS
000005                                THE REGISTERS IT IS GIVEN WITH THE
000005                                VALUES IN THE STACK NODE *)
000005          (* GLOVAR STACK; GLOBAL PROCEDURE RECYCLE *)
000005          BEGIN
000005          ENODE:= MEMORY[STACK];
000014          RECYCLE(REG1);
000020          RECYCLE(REG2);
000025          REG1:= ENODE.REF.RR;  (* USE THE STACKS NUDGE *)
000032          REG2:= ENODE.CAR.RR;
000036          DISPOZE(STACK);
000041          STACK:= ENODE.CDR.RR;
000046          IF (TRACE>5) THEN
000050              WRITELN("      LOAD... REG1 IS ",REG1:1,", REG2 IS ",REG2:1,"
000074          END;    (* END OF PROCEDURE LOAD *)
000110
000110
000110      PROCEDURE EVALERROR(I: INTEGER;MESSAGE:PTR);
000005                             (* A NOTICIBLE ERROR CAUSES A RECOVERY
000005                                OF THE STACK TO AVAILABLE SPACE. *)
000005              (* GLOVAR STACK, AVAIL; GLOCON NEWSUSPEND *)
000005          LABEL 1;
000005          BEGIN
000005          RECYCLE(STACK);
000014          STACK:= NIL;
000016          IF (I=0) THEN GOTO 1;
000020          WRITELN;
000021          WRITE("-=>-=> EVALUATION ERROR:  ");
000026          CASE I OF
000033          1: WRITE("UNDEFINED FUNCTION,");
000041          2: WRITE("UNMARKED FUNARG,");
000047          3: WRITE("SECOND ORDER NOT PERMITTED YET.");
```

```
        4: WRITE("NON-NUMERIC ARGUMENT,");
        5: WRITE("TOO FEW ARGUMENTS,");
        6: WRITE("NON-POSITIVE NUMERIC,");
        7: WRITE("STRUCTURE MATCH FAILED,");
        8: WRITE("NO CONVERGENCE IN LIST.");
        9: WRITE("CAR APPLIED TO");
        10: WRITE("CDR APPLIED TO");
        18: WRITE("REDIFINED CONSTANT,");
        19: WRITE("UNBOUND VARIABLE,");
        END;  (* CASE *)
        WRITE(" ");
        IF ISATOM(MESSAGE) THEN PUTAT(MESSAGE)
        ELSE WRITE("
);
        WRITELN;
    1:  SETREG(REVAL,JAWS)
        END;   (* END OF PROCEDURE EVALERROR *)


    PROCEDURE CONTEXTPUSH(NODEP:PTR;ISCAR:BOOLEAN; RESOURCES:INTEGER);
        (* THE STAQUE IS A LIST OF CONTINUATIONS.  WHEN A SUSPENSION
           IS ENCOUNTERED DURING A LIST PROBE, PUSH THE CURRENT PRO-
           CESS ONTO STAQUE AND START PROCESSING THE SUSPENSION *)
        LABEL 1;
        VAR PT:PTR; BUSY:REFERENCE; CNODE:NODE;
        BEGIN
        (* NODEP IS CONS-MULTI AND HAS BEEN RESERVED *)
        PUSHONE(RESTORE,ENVDOT);
        IF ISCAR THEN BUSY:= MEMORY[NODEP].CAR
        ELSE BUSY:= MEMORY[NODEP].CDR;
        IF BUSY.NUMBERP THEN
            BEGIN
            MODECODE:= 0;
            GOTO 1
            END;
        CNODE:= NEWSUSPEND;
        CNODE.CDR.RR:= STAQUE;
        CNODE.CAR.RR:= NODEP; NUDGE(NODEP);
        CNODE.REF.RR:= STACK;
        CNODE.PNAME:= ISCAR;
        STAQUE:= NEWNODE; MEMORY[STAQUE]:= CNODE;
        BUSY.NUMBERP:= TRUE;
        BUSY.NN:= MODECODE;
        MODECODE:= RESOURCES;
        IF (TRACE>3) THEN
            BEGIN
            WRITELN; WRITE(" CONTEXT PUSH: ",STAQUE:1," --> ");
            WRITENODE(STAQUE,FALSE);
            IF ISCAR THEN WRITELN(", PROCESS-CAR.")
            ELSE WRITELN(", PROCESS-CDR.");
            END;
        IF ISCAR THEN
            BEGIN
            STACK:= MEMORY[NODEP].CAR.RR;
            MEMORY[NODEP].CAR:= BUSY
            END
        ELSE
            BEGIN
            STACK:= MEMORY[NODEP].CDR.RR;
```

```
000201              MEMORY[NODEP].CDR:= BUSY;
000206              END;
000206     1:   END;
000230
000230
000230
000230     PROCEDURE CONTEXTPOP(VALUE:PTR);
000004          (* A POP OF THE CONTINUATION STACK *)
000004          VAR PT:PTR; CNODE:NODE;
000006          BEGIN
000006          CNODE:= MEMORY[STAQUE];
000013          IF (TRACE>3) THEN
000015              BEGIN
000015              WRITELN;WRITE("CONTEXT POP: ");
000023              WRITENODE(STAQUE,FALSE);
000032              WRITE("; FILL-");
000037              IF CNODE.PNAME THEN WRITELN("CAR.") ELSE WRITELN("CDR.");
000056              END;
000056          DISPOZE(STAQUE);
000063          PT:= CNODE.CAR.RR;
000067          IF CNODE.PNAME THEN    (* THE VALUE GOES IN CAR *)
000071              BEGIN
000071              MODECODE:= MEMORY[PT].CAR.NN;
000075              MEMORY[PT].CAR.NUMBERP:= FALSE;
000102              MEMORY[PT].CAR.RR:= VALUE
000104              END
000112          ELSE
000112              BEGIN
000112              MODECODE:= MEMORY[PT].CDR.NN;
000117              MEMORY[PT].CDR.NUMBERP:= FALSE;
000124              MEMORY[PT].CDR.RR:= VALUE
000125              END;
000132          NUDGE(VALUE);
000136          RECYCLE(PT);
000142          STACK:= CNODE.REF.RR;
000147          STAQUE:= CNODE.CDR.RR
000147          END;
000166
000166
000166     FUNCTION ALLOCATE(STRATEGY:INTEGER):INTEGER;
000004                              (* 'CAR', 'CDR', 'KICKAR', OR 'KICKDR' IS
000004                                 DOING A PROCESS SWAP.  THIS FUNCTION
000004                                 DISTRIBUTES THE RESOURCES.  NOTE THE
000004                                 SIDEEFFECT ON THE EVAL-LOCAL MODECODE *)
000004          VAR N:INTEGER;
000005          BEGIN
000005          IF (MODECODE>=INFINITY) THEN N:= INFINITY
000011          ELSE IF (MODECODE=0) THEN N:= 0
000013          ELSE IF (STRATEGY=1) THEN
000015              BEGIN
000015              N:= (MODECODE DIV 2) +1;
000017              MODECODE:= MODECODE-N
000017              END
000017          ELSE IF (STRATEGY=2) THEN
000021              BEGIN
000021              N:= (MODECODE DIV 3) + 1;
000027              MODECODE:= MODECODE-N
000027              END
```

```
        ELSE IF (STRATEGY=3) THEN
            BEGIN
            N:= 1;
            MODECODE:= MODECODE-1
            END;
        ALLOCATE:= N
        END;


    PROCEDURE KICKLIS(LIST:PTR);
                            (* ONE OF THE POSSIBLE EVALUATION
                               STRATEGIES FOR ORDER-BY-CONVERGENCE
                               OF MULTISETS.  LOOK FOR A VALUE, AND
                               MOVE IT TO THE BEGINNING OF THE
                               LIST.  IF THERE IS NONE, KICK
                               ALL SUSPENSIONS.  STOP AT NIL,
                               A SUSPENDED CDR, OR A CONS    *)
        LABEL 1,2,3,4,5;
        VAR PT,PT1,PT2,HIT:PTR; KNODE:NODE; NOHOPE:BOOLEAN;
        BEGIN
        NOHOPE:= (MEMORY[LIST].CAR.RR=JAWS);
        PT:= LIST;
        PT1:= NIL;
        PUSHONE(CAR,LIST);
        (* LOOK FOR A VALUE, REVERSING AS YOU GO. *)
        IF SUSPENDED(MEMORY[PT].CDR.RR) THEN
            BEGIN
            NOHOPE:= FALSE;
            GOTO 1
            END;
        REPEAT
            PT2:= PT;
            PT:= MEMORY[PT].CDR.RR; KNODE:= MEMORY[PT];
            MEMORY[PT2].CDR.RR:= PT1;
            PT1:= PT2;
            IF (* RESERVED(PT) OR *) ISATOM(PT) THEN GOTO 3;
            IF NOT (KNODE.CAR.RR=JAWS) THEN
                BEGIN
                NOHOPE:= FALSE;
                IF NOT SUSPENDED(KNODE.CAR.RR) THEN GOTO 4;
                IF NOT KNODE.MULTI THEN GOTO 2;
                IF SUSPENDED(KNODE.CDR.RR) THEN GOTO 1
                END;
        UNTIL FALSE;
    1:  PUSHONE(KICKDR,PT);
    2:  PUSHONE(KICKAR,PT);
    3:  (** CANCEL(PT); **)
        (* HERE NO VALUE HAS BEEN FOUND.  TRAVERSE-REVERSE TO
           GET THINGS BACK AS THEY WERE. *)
        WHILE NOT (PT1=NIL) DO
            BEGIN
            PT2:= PT1;
            PT1:= MEMORY[PT1].CDR.RR;
            MEMORY[PT2].CDR.RR:= PT;
            PT:= PT2;
            PUSHONE(KICKAR,PT);
            (** CANCEL(PT) **)
            END;
```

```
          IF NOHOPE THEN EVALERROR(O,NIL);
          GOTO 5;
      4:  (** CANCEL(PT); **)
          HIT:= PT;
          POP;  (* THE PUSH AT BEGINNING OF THIS PROCEDURE *)
          SETREG(REVAL,MEMORY[HIT].CAR.RR);
          PT:= PT1;
          PT1:= MEMORY[PT1].CDR.RR;
          PT2:= MEMORY[HIT].CDR.RR;
          IF SUSPENDED(PT2) THEN  (* IMPOSE AN INTERMEDIATE SUSPENSION *)
              BEGIN
              PT2:= DOTPAIR(FCDR,HIT);
              KNODE:= NEWSUSPEND;
              KNODE.REF.RR:= PT2; NUDGE(PT2);
              KNODE.CAR.NUMBERP:= TRUE;
              KNODE.CAR.NN:= TOP;
              PT2:= NEWNODE;
              MEMORY[PT2]:= KNODE
              END;
          REPEAT
              MEMORY[PT].REF.NN:= MEMORY[PT].REF.NN-1;
              IF MEMORY[PT].REF.NN=0 THEN
                  BEGIN
                  MEMORY[PT].CDR.RR:= PT2; NUDGE(PT2);
                  PT2:= PT
                  END
              ELSE
                  BEGIN
                  KNODE:= NEWCONS;
                  KNODE.MULTI:= TRUE;
                  KNODE.REF.NN:= 1;
                  KNODE.CAR:= MEMORY[PT].CAR;
                  KNODE.CDR.RR:= PT2; NUDGE(PT2);
                  PT2:= NEWNODE; MEMORY[PT2]:= KNODE;
                  MEMORY[PT].CAR.RR:= REVAL; NUDGE(REVAL);
                  MEMORY[PT].CDR.RR:= PT2
                  END;
              (** CANCEL(PT) **)
              PT:= PT1;
              PT1:= MEMORY[PT1].CDR.RR;
          UNTIL PT=NIL;
          NUDGE(LIST);
          RECYCLE(HIT);
    5:    END;  (* PROCEDURE KICKLIS *)






PROCEDURE APPLY(F,A:PTR);
                          (* CALLED FROM EVAL.  F IS A NUMBER,
                             ATOM, OR LIST; A IS THE EVALUATED
                             ARGUMENT *)
    LABEL 1;       (*"LAMBDA AND ?FUNARG => A SECOND CALL TO APPLY*)
    VAR PT,PT1,TEM : PTR; (* INSPECTION-REGISTERS *)
        (* FN AND ARGS ARE THE VALUE-REGISTERS *)
    BEGIN
```

```
000010     SETREG(FN,F); SETREG(ARGS,A);
000020 1:  IF (TRACE>3) THEN WRITELN("       APPLY ",FN:1," TO ",ARGS:1);
000045     IF (FN=NIL) THEN
000047          SETREG(REVAL,NIL)
000052     ELSE IF MEMORY[FN].CAR.NUMBERP THEN
000060          BEGIN
000060          N:= MEMORY[FN].CAR.NN;
000064          IF (N<1) THEN EVALERROR(6,FN)
000070          ELSE IF (N=1) THEN PUSHONE(CAR,ARGS)
000077          ELSE
000101              BEGIN
000101              PUSHONE(N-1,NIL);
000105              PUSHONE(NTH,NIL);
000112              PUSHONE(CDR,ARGS)
000115              END
000117          END
000117     ELSE IF MEMORY[FN].ATOMP THEN    (* FN IS ATOMIC *)
000124          BEGIN
000124          IF FN = QCAR THEN PUSHONE(CAR,ARGS)
000131             (*MOD3 BEGIN
000131             PUSHONE(ACAR,NIL);
000131             PUSHONE(CAR,ARGS)
000131             END3DOM*)
000131          ELSE IF (FN = QCDR) THEN PUSHONE(CDR,ARGS)
000140             (*MOD3BEGIN
000140             PUSHONE(ACDR,NIL);
000140             PUSHONE(CAR,ARGS)
000140             END3DOM*)
000140          ELSE IF (FN = QCONS) OR (FN = QFONS) THEN
000146             BEGIN
000146             PT:= DOTPAIR(FCAR,ARGS);
000160             PT1:= DOTPAIR(FCADR,ARGS);
000172             SETREG(REVAL,CONS(PT,PT1,NIL));
000211             MEMORY[REVAL].MULTI:= (FN=QFONS)
000215             END
000223          ELSE IF (FN = QSTAR) THEN
000226             BEGIN
000226             PT:= DOTPAIR(QARGS,NIL);
000237             PT:= DOTPAIR(QCAR,PT);     (* PT: (QCAR QARGS)   *)
000251             PT1:= MAKEENV(ARGS);       (* SEE CONS  *)
000260             SETREG(REVAL,STAR(PT,PT1))
000274             END
000275          ELSE IF (FN = QSTARRED) THEN
000300             BEGIN
000300             PUSHONE(ASTAR,NIL);
000305             PUSHONE(CAR,ARGS)
000310             END
000312          ELSE IF (FN=QATOM) OR (FN=QADD1) OR (FN=QSUB1) OR
000321                  (FN=QNULL) OR (FN=QNOT)  THEN
000325             BEGIN
000325             PUSHONE(ADD1,FN);
000331             SETREG(REVAL,ARGS)
000334             END
000335          ELSE IF (FN = QHSH) THEN
000340             PUSHONE(CAR,ARGS)
000343          ELSE IF ((FN=QPLUS) OR (FN=QDIFF) OR (FN=QTIMES) OR  (FN=QMOD) OR
000355                    (FN=QDIV) OR (FN=QLESS) OR (FN=QGREAT) OR (FN=QEQ))   TH
000364              BEGIN
```

```
000364              IF (FN=QEQ) THEN PUSHONE(EQ,ARGS)
000370              ELSE
000373                  BEGIN
000373                  PUSHTWO(FN,ARGS);
000402                  PUSHONE(PLUS,NIL)
000405                  END;
000407              PUSHONE(CAR,ARGS);
000414              PUSHONE(ACAR,NIL);
000421              PUSHONE(CDR,ARGS)    (* COERCE THE CAR AND CADR FOR EVAL *)
000424              END
000426          ELSE        (*INSERT OTHER COMPILED FUNCTIONS HERE
000427                        SEARCH FOR USER DEFINED FUNCTION *)
000427             BEGIN
000427             SETREG(REVAL,SEARCH(FLIST,FN));
000445             IF MEMORY[REVAL].MULTI THEN
000452                  BEGIN
000452                  SETREG(FN,CADR(REVAL));
000464                  GOTO 1
000465                  END;
000465             PUSHTWO(FN,ARGS);
000474             PUSHONE(CHECK1,NIL);
000501             PUSHONE(RESTORE,ENVDOT);
000506             SETREG(REVAL,QUNBOUND);
000512             PUSHONE(ASSOC,FN)
000515             END
000517          END          (* THIS ENDS CASES WHEN FN IS AN ATOM *)
000517     ELSE              (* THIS STARTS CASES WHEN FN IS A LIST *)
000520          BEGIN
000520          PT1:= MEMORY[FN].CAR.RR;
000526          IF (PT1 = QLAMBDA) THEN
000530             BEGIN
000530             PUSHONE(RESTORE,ENVDOT);
000534             SETREG(ENVDOT,DOTPAIR(DOTPAIR(CADR(FN),ARGS),NIL));
000563                  (* FOR FREE VARIABLES, EXCHANGE ENVDOT FOR NIL *)
000563             PUSHONE(COND,CDDR(FN))
000575             END
000576          ELSE IF (PT1 = QFUNARG) THEN
000600             BEGIN
000600             PUSHONE(RESTORE,ENVDOT);
000605             SETREG(ENVDOT,CADDR(FN));
000620             SETREG(FN,CADR(FN));
000633             GOTO 1                   (* NEW ENVIRONMENT, NOW APPLY  *)
000634             END
000634          (* ELSE IF (PT1 = QLABEL) THEN
000634             BEGIN
000634             A SEMANTICS FOR 'LABEL' HAS NOT BEEN ESTABLISHED.;
000634             GOTO 1
000634             END *)
000634          ELSE
000634             BEGIN
000634             PUSHTWO(FN,ARGS);
000642             PUSHONE(FNSTAR,NIL);    (* FNCTNL COMBINATION. CHECK FOR STARS
000647             PUSHONE(ASTAR,NIL);   (* SEE IF THE F-C IS STARRED *)
000654             SETREG(REVAL,FN)
000657             END
000660          END
000660     END;    (* THE PROCEDURE APPLY *)
000700
```

```
000700
000700
000700
000700
000700
000700
000700 BEGIN (* BEGIN THE PROCEDURE EVALHELP *)
000700 MODECODE:= RESOURCES;
000010 STAQUE:= NIL;
000011 ALLDONE:= FALSE;
000012 IF (TRACE>5) THEN
000014      BEGIN
000014      WRITELN;
000015      WRITELN("-=>-=>-=> EVALUATION. PROCESS STACK IS NODE ",STACK:1,".");
000031      WRITELN
000031      END;
000032 POP;
000034 WHILE NOT ALLDONE DO
000036 BEGIN
000036 CASE PLACE OF
000042    TOP:
000042          BEGIN
000042          IF (TRACE>3) THEN WRITELN("TOP  ");
000052          IF (EXP=NIL) OR (EXP=QFALSE) THEN
000056               SETREG(REVAL,NIL)
000062          ELSE IF MEMORY[EXP].CAR.NUMBERP THEN
000070               SETREG(REVAL,EXP)
000073          ELSE IF MEMORY[EXP].ATOMP THEN  (*THE EXPRESSION IS AN ATOM*)
000101               BEGIN
000101               IF (EXP=QHSH) THEN SETREG(REVAL,EXP)
000105               ELSE IF (EXP=JAWS) THEN EVALERROR(0,NIL)
000113               ELSE IF EXP=QSPEAK THEN SETREG(REVAL,MAKENUM(SPEAK))
000130               ELSE IF EXP=QSTOP THEN
000133                    BEGIN
000133                    ALLDONE:= TRUE;
000134                    FINIS:= TRUE
000134                    END
000135               ELSE IF EXP=QENV THEN SETREG(REVAL,ENVDOT)
000142              ELSE IF EXP=QUNDEFINED THEN SETREG(REVAL,JAWS)
000151               (* INSERT OTHER SYSTEM ATOMS HERE *)
000151               ELSE
000153                    BEGIN    (* SEARCH THE ENVIRONMENT FOR A BINDING *)
000153                    PUSHONE(RESTORE,ENVDOT);  (* ASSOC DISECTS ENVDOT *)
000160                    SETREG(REVAL,QUNBOUND);   (* INITIALIZE ASSOC *)
000164                    PUSHONE(LOOK,EXP);        (* IN CASE OF ERROR *)
000171                    PUSHONE(ASSOC,EXP)
000174                    END
000176               END
000176          ELSE
000177               BEGIN   (* THE EXPRESSION IS A LIST.  *)
000177               IF EXP=MEMORY[EXP].CDR.RR THEN
000203                    BEGIN           (* THE EXPRESSION IS STARRED *)
000203                    PUSHONE(STARRED,EXP);
000207                    PUSHONE(CAR,EXP)
000212                    END
000214               ELSE
000215                    BEGIN
000215                    PT:= MEMORY[EXP].CAR.RR;  (* GRAB THE FUNCTION *)
```

```
000223                  PT1:= MEMORY[EXP].CDR.RR;
000230                  IF ISATOM(PT) THEN
000235                      BEGIN
000235                      IF (PT=QTRACE) THEN
000237                          BEGIN
000237                          SETREG(REVAL,EXP);
000243                          TRACE:= MEMORY[MEMORY[PT1].CAR.RR].CAR.NN
000251                          END
000253                      ELSE IF (PT=QEVCAR) THEN
000255                          BEGIN
000255                          IF SUSPENDED(CADR(EXP)) THEN
000266                              BEGIN
000266                              PUSHONE(TOP,EXP);
000272                              PUSHONE(CAR,MEMORY[EXP].CDR.RR)
000301                              END
000302                          ELSE
000303                              PUSHONE(TOP,CADR(EXP))
000314                          END
000315                      ELSE IF (PT=QCARLIS) THEN PUSHONE(CARLIS,PT1)
000323                      ELSE IF (PT=FCAR) THEN PUSHONE(CAR,PT1)
000332                      ELSE IF (PT=FCDR) THEN PUSHONE(CDR,PT1)
000341                      ELSE IF (PT=FCADR) THEN
000344                          BEGIN
000344                          PUSHONE(ACAR,NIL);
000351                          PUSHONE(CDR,PT1)
000354                          END
000356                      ELSE IF (PT=FCDAR) THEN
000360                          BEGIN
000360                          PUSHONE(ACDR,NIL);
000365                          PUSHONE(CAR,PT1)
000370                          END
000372                      ELSE IF (PT=QCDRLIS) THEN PUSHONE(CDRLIS,PT1)
000400                      ELSE IF (PT=FAPPLY)  THEN
000404                          BEGIN
000404                          PUSHONE(RESTORE,ENVDOT);
000410                          SETREG(ENVDOT,CADDDR(EXP));
000423                          APPLY(CADR(EXP),CADDR(EXP))
000442                          END
000443                      ELSE IF (PT=QCONS) OR (PT=QFONS) THEN
000450                          BEGIN
000450                          SETREG(REVAL,CONS(CADR(EXP),CADDR(EXP),ENVDOT))
000500                          MEMORY[REVAL].MULTI:= (PT=QFONS)
000504                          END
000512                      ELSE IF (PT=QSTAR) THEN
000515                          SETREG(REVAL,STAR(CADR(EXP),ENVDOT))
000534                      ELSE IF (PT = QCOLON) THEN
000540                          BEGIN
000540                          PUSHONE(ANB,CADR(EXP));
000552                          PUSHONE(TOP,CADDR(EXP))
000563                          END
000564                      ELSE IF (PT=QQUOTE) THEN
000567                          SETREG(REVAL,CADR(EXP))
000600                      ELSE IF (PT=QFUNCTIO) THEN
000603                          BEGIN
000603                          POINT:= DOTPAIR(ENVDOT,NIL);
000615                          POINT:= DOTPAIR(CADR(EXP),POINT);
000632                          SETREG(REVAL,DOTPAIR(QFUNARG,POINT))
000646                          END
```

```
000647                  ELSE IF (PT=QDE) THEN
000652                      BEGIN
000652                      PT1:= DOTPAIR(QLAMBDA,CDDR(EXP));
000671                      SETREG(REVAL,SEARCH(FLIST,CADR(EXP)));
000714                      IF MEMORY[REVAL].MULTI THEN
000721                          BEGIN
000721                          RECYCLE(MEMORY[MEMORY[REVAL].CDR.RR].CAR.RR);
000732                          MEMORY[MEMORY[REVAL].CDR.RR].CAR.RR:= PT1;
000746                          NUDGE(PT1);
000750                          PT:= DOTPAIR(QREDEF,NIL);
000762                          SETREG(REVAL,DOTPAIR(CADR(EXP),PT))
001001                          END
001002                      ELSE
001003                          BEGIN
001003                          PT:= DOTPAIR(CADR(EXP),NIL);
001020                          MEMORY[MEMORY[REVAL].CAR.RR].CDR.RR:= PT;
001031                          NUDGE(PT);
001034                          PT:= DOTPAIR(PT1,NIL);
001046                          MEMORY[MEMORY[REVAL].CDR.RR].CDR.RR:= PT;
001057                          NUDGE(PT);
001062                          SETREG(REVAL,CADR(EXP))
001074                          END
001075                      END
001075                  ELSE IF (PT=QMLIST) OR (PT=QCLIST) THEN
001102                      PUSHONE(EVLIS,MEMORY[EXP].CDR.RR)
001111                  ELSE IF (PT=QDC) THEN
001114                      BEGIN
001114                      PUSHONE(DE,MEMORY[PT1].CAR.RR);
001124                      PUSHONE(TOP,CADR(PT1))
001135                      END
001136                  ELSE
001137                      BEGIN
001137                      PUSHONE(ANB,PT);
001143                      PUSHONE(EVLIS,MEMORY[EXP].CDR.RR)
001152                      END
001153                  END
001153              ELSE
001154                  BEGIN
001154                  PUSHONE(ANB,FN);
001161                  PUSHONE(EVLIS,MEMORY[EXP].CDR.RR)
001170                  END
001171              END
001171          END
001171      END;
001172
001172
001172  RESTORE:        (* THIS CELL OF THE STACK HAS HELD ONTO
001172                      THE ENVIRONMENT IN EFFECT DURING A PREVIOUS
001172                      CALL TO EVAL.  IT CHANGES THE ENVIRONMENT
001172                      REGISTER BACK TO THAT VALUE.
001172                         NOTE THAT THIS VERSION OF THE IMPLEMENTATION
001172                      WASTES SEVERAL ASSIGNMENT CALLS TO SAVE
001172                      THIS LIST.  WHEN OPTIMIZING, THE CALL TO
001172                      SETREG CAN BE REPLACED BY A MACRO THAT
001172                      DOES THE PUSH WITH NO NUDGES. *)
001172      BEGIN
001172      IF (TRACE>3) THEN WRITELN('RESTORE ');
001202      SETREG(ENVDOT,EXP)
```

```
     END;

LOOK:        (* SEE IF VARIABLE SEARCH FAILD. *)
     BEGIN
     IF (TRACE>3) THEN WRITELN("LOOK ");
     IF (REVAL=QUNBOUND) THEN EVALERROR(19,EXP)
     END;

ASSOC:      (*  THE ENVIRONMENT IS A LIST OF
                     FORMAL PARAMETER-ARGUMENT PAIRS.
                     SEARCH EACH PAIR FOR A BINDING.
                     EXP IS THE VARIABLE NAME.
                     THE CALLING ROUTINE MUST SAVE THE
                     ENVIRONMENT. *)
     BEGIN
     IF (TRACE>3) THEN WRITELN("ASSOC ");
     IF (REVAL=QUNBOUND) THEN
          BEGIN
          IF (ENVDOT=NIL) THEN
               BEGIN
               SETREG(EXP,SEARCH(ALIST,EXP));
               IF MEMORY[EXP].MULTI THEN SETREG(REVAL,CADR(EXP))
               END
          ELSE
               BEGIN
               PUSHONE(ASSOC,EXP);
               PUSHTWO(EXP,CAAR(ENVDOT));
               PUSHONE(ASSOC1,MEMORY[MEMORY[ENVDOT].CAR.RR].CDR.RR);
               SETREG(ENVDOT,MEMORY[ENVDOT].CDR.RR)
               END
          END
     END;

ASSOC1:
     BEGIN
     IF (TRACE>3) THEN WRITELN("ASSOC1: ");
     LOAD(ASSOCVAR,FP);     (* EXP IS THE ACTUAL PARAMETER LIST *)
     IF (FP = ASSOCVAR) THEN SETREG(REVAL,EXP)
     ELSE IF ISATOM(FP) THEN SETREG(REVAL,QUNBOUND)
     ELSE IF ISATOM(EXP) THEN EVALERROR(7,ASSOCVAR)
     ELSE
          BEGIN
          PT:= MEMORY[FP].CAR.RR;
          IF (PT = ASSOCVAR) THEN PUSHONE(CAR,EXP)
          ELSE IF ISATOM(PT) THEN
               BEGIN
               PUSHTWO(ASSOCVAR,MEMORY[FP].CDR.RR);
               PUSHONE(ASSOC2,NIL);
               PUSHONE(CDR,EXP)       (* COERCE, AND CONTINUE SEARCH *)
               END
```

```
          ELSE   (* FIRST:FP IS A LIST, RECUR ON CAR *)
              BEGIN
              PUSHTWO(ASSOCVAR,MEMORY[FP].CDR.RR);
              PUSHONE(ASSOC3,EXP);
              PUSHTWO(ASSOCVAR,MEMORY[FP].CAR.RR);
              PUSHONE(ASSOC2,NIL);
              PUSHONE(CAR,EXP)
              END
          END
      END;



ASSOC2:          (* CDR OF ACTUAL PARAMETERS IS MANIFEST,
                      CONTINUE ASSOCIATION SEARCH *)
      BEGIN
      IF (TRACE>3) THEN WRITELN("ASSOC2 ");
      (* VARIABLE AND FORMAL PARAMETERS ARE ON STACK *)
      PUSHONE(ASSOC1,REVAL)
      END;



ASSOC3:           (* ASSOC HAS RECURED ON FIRST:FP.
                      IF SEARCH FAILED RECUR ON REST:FP *)
      BEGIN
      IF (TRACE>3) THEN WRITELN("ASSOC3 ");
      IF (REVAL=QUNBOUND) THEN
          BEGIN
          PUSHONE(ASSOC2,NIL);
          PUSHONE(CDR,EXP)
          END
      ELSE
          LOAD(ASSOCVAR,FP)
      END;

KICKAR:     (* EXPEND ONE UNIT OF RESOURCE ON EVALUATING
                      THE CAR FIELD OF 'EXP' *)
      BEGIN
      IF (TRACE>3) THEN WRITELN("KICKAR ");
      (** IF RESERVED(EXP) THEN BEGIN END
      ELSE IF MEMORY[EXP].CAR.NUMBERP THEN
          MEMORY[EXP].CAR.NN:= MEMORY[EXP].CAR.NN+1
      ELSE **)
      IF SUSPENDED(MEMORY[EXP].CAR.RR) THEN
          CONTEXTPUSH(EXP,TRUE,ALLOCATE(3));
      (** CANCEL(EXP) **)
      END;



KICKDR:  (* SAME AS KICKAR BUT FOR CDR *)
      BEGIN
      IF (TRACE>3) THEN WRITELN("KICKDR ");
      (** RESERVE NODE; ADD RESOURCES TO PROCESSOR
          ALREADY WORKING ON CDR.  SEE KICKAR **)
      IF SUSPENDED(MEMORY[EXP].CDR.RR) THEN
          CONTEXTPUSH(EXP,FALSE,ALLOCATE(3))
```

```
001672        END;
001674
001674
001674
001674    CAR:       (*THIS IS THE USER CAR.  EXP HAS THE NODE
001674                     WHOSE CAR IS TO BE RETURNED.  CHECK
001674                     FOR A SUSPENSION.  RETURN THE CAR
001674                     IF IT ISNT SUSPENDED, ELSE EVALUATE IT. *)
001674        BEGIN
001674        IF (TRACE>3) THEN WRITELN("CAR ");
001704        IF ISATOM(EXP) THEN EVALERROR(9,EXP)
001714        (** ELSE IF RESERVED(EXP) THEN
001714            BEGIN
001714            PUSHONE(CAR,EXP);
001714            MODECODE:= MODECODE-1
001714            END **)
001714        ELSE
001716            BEGIN
001716            PT:= MEMORY[EXP].CAR.RR;
001725            IF (PT=JAWS) THEN
001727                BEGIN
001727                IF MEMORY[EXP].MULTI THEN KICKLIS(EXP)
001734                ELSE EVALERROR(0,NIL)
001740                END
001742            ELSE IF NOT SUSPENDED(PT) THEN
001750                BEGIN
001750                SETREG(REVAL,PT);
001754                (** CANCEL(EXP) **)
001754                END
001754            ELSE IF NOT MEMORY[EXP].MULTI THEN
001762                BEGIN
001762                PUSHONE(CAR,EXP);
001766                CONTEXTPUSH(EXP,TRUE,ALLOCATE(1));
002001                (** CANCEL(EXP) **)
002001                END
002001            ELSE
002002                BEGIN
002002                IF (TRACE>3) THEN WRITELN("-=>-=>-=> MULTI <=-<=-<=-");
002012                KICKLIS(EXP)
002015                END
002016            END
002016        END;
002017
002017
002017    CDR:    (* THE USERS CDR.  SEE THE NOTE ON CAR, JUST ABOVE *)
002017        BEGIN
002017        IF (TRACE>3) THEN WRITELN("CDR ");
002027        IF ISATOM(EXP) THEN EVALERROR(10,EXP)
002037        (** ELSE IF RESERVED(EXP) THEN
002037            BEGIN
002037            PUSHONE(CDR,EXP);
002037            MODECODE:= MODECODE-1
002037            END **)
002037        ELSE
002041            BEGIN
002041            PT:= MEMORY[EXP].CAR.RR;
002050            PT1:= MEMORY[EXP].CDR.RR;
002055            IF MEMORY[EXP].MULTI AND SUSPENDED(PT)  THEN
```

```
002067                 BEGIN
002067                 IF (TRACE>3) THEN WRITELN("-=>-=>-=> MULTI <=-<=-<=- ");
002077                 PUSHONE(CDR,EXP);
002104                 PUSHONE(CAR,EXP)
002107                 END
002111             ELSE IF SUSPENDED(PT1) THEN
002117                 BEGIN
002117                 PUSHONE(CDR,EXP);
002123                 CONTEXTPUSH(EXP,FALSE,ALLOCATE(2))
002135                 END
002136             ELSE
002137                 SETREG(REVAL,PT1);
002143             (** CANCEL(EXP) **)
002143             END
002143         END;
002144
002144
002144
002144
002144     STARRED:
002144         BEGIN
002144         IF (TRACE>3) THEN WRITELN("STARRED ");
002154         SETREG(REVAL,STAR(REVAL,ENVDOT))
002171         END;
002173
002173
002173     COND:           (* THE LISP CONDITIONAL.  THE KEYWORDS
002173                         =IF=, =THEN=, =ELSEIF=, AND =ELSE= ARE
002173                         IGNORED WHEN THEY ARE ENCOUNTERED. THIS
002173                         IS THE BOTTOM LINE.  RETURN NIL IF THE
002173                         LIST OF PAIRS IS EMPTY.  IF THERE IS
002173                         THE LAST ARGUMENT IS NOT A PAIR, RETURN
002173                         IT.  OTHERWISE EVALUATE THE PREDICATE
002173                         PART OF THE PAIR.
002173                           EXAMPLE:
002173                              IF <NULL [PRS]> THEN NIL
002173                              IF <MEMBER [<CAR [PRS]> ?(IF THEN ELSEIF EL
002173                                THEN <COND [<CDR [PRS]>]>
002173                              ETC...    *)
002173         BEGIN
002173         IF (TRACE>3) THEN WRITELN("COND ");
002203         IF EXP=NIL THEN SETREG(REVAL,NIL)      (* EXP IS THE LIST OF PAIRS
002210         ELSE
002212             BEGIN
002212             PT:= MEMORY[EXP].CAR.RR;
002221             IF ((PT=QIF)OR(PT=QTHEN))OR((PT=QELSEIF)OR(PT=QELSE)) THEN
002230                 PUSHONE(COND,MEMORY[EXP].CDR.RR)
002237             ELSE IF MEMORY[EXP].CDR.RR=NIL THEN
002246                 PUSHONE(TOP,PT)
002251             ELSE
002253                 BEGIN
002253                 PUSHONE(COND1,MEMORY[EXP].CDR.RR);
002263                 PUSHONE(TOP,PT)
002266                 END
002270             END
002270         END;
002271
002271
```

```
    COND1:                  (* REVAL HAS THE PREDICATE PART OF A COND
                                  PAIR.  EXP HAS THE REST OF THE COND LIST.
                                  AGAIN, SKIP HELP-WORDS IF, THEN ETC. *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("COND1 ");
        IF EXP=NIL THEN BEGIN END (* THIS SHOULD NEVER HAPPEN *)
        ELSE
            BEGIN
            PT:= MEMORY[EXP].CAR.RR;
            IF ((PT=QIF)OR(PT=QTHEN))OR((PT=QELSEIF)OR(PT=QELSE)) THEN
                PUSHONE(COND1,MEMORY[EXP].CDR.RR)
            ELSE IF REVAL=JAWS THEN EVALERROR(0,NIL)
                        (* IMPORTANT... THE SEMANTICS OF
                           CONDITIONAL STATEMENTS MAY BE
                           CHANGED TO HANDLE THIS CASE.
                           SEE SECTION 2.8.*)
            ELSE IF REVAL=NIL THEN
                PUSHONE(COND,MEMORY[EXP].CDR.RR)
            ELSE
                PUSHONE(TOP,PT)
            END
        END;


    ANB:               (* REVAL IS THE EVALUATED ARGUMENT *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("ANB ");
        APPLY(EXP,REVAL)
        END;


    DE:            (* CONSTANT DECLARATION.*)
        BEGIN
        IF (TRACE>3) THEN WRITELN("DE ");
        SETREG(AP,SEARCH(ALIST,EXP));
        IF MEMORY[AP].MULTI THEN EVALERROR(18,EXP)
        ELSE
            BEGIN
            PT:= DOTPAIR(EXP,NIL); NUDGE(PT);
            MEMORY[MEMORY[AP].CAR.RR].CDR.RR:= PT;
            PT:= DOTPAIR(REVAL,NIL); NUDGE(PT);
            MEMORY[MEMORY[AP].CDR.RR].CDR.RR:= PT;
            END
        END;


    CHECK1:                 (* AN ASSOC-SEARCH WAS JUST DONE.  IF IT
                               TURNED UP NOTHING,THEN UNDEFINED FUNCTION.
                               APPLY THE RESULT OF THE SEARCH TO THE ARGS. *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("CHECK1 ");
        LOAD(FN,ARGS);
        IF (REVAL=QUNBOUND) THEN EVALERROR(1,FN)
        ELSE
            BEGIN
            IF MEMORY[REVAL].CAR.NUMBERP THEN APPLY(REVAL,ARGS)
            ELSE
                BEGIN
```

```
                PUSHTWO(REVAL,ARGS);
                PUSHONE(CHECK2,NIL);
                PUSHONE(CAR,REVAL);    (* LOOK FOR 'FUNARG *)
                END
            END
        END;


    CHECK2:     (* WE ARE ABOUT TO APPLY.  LOOKING FOR A FUNARG.
                        REVAL IS (CAR (CDR (ASSOC FN ENVIRONMENT))) *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("TEST2 ");
        LOAD(FN,ARGS);       (* EXP IS NIL *)
        IF (REVAL=QFUNARG) THEN APPLY(FN,ARGS)
        ELSE EVALERROR(2,REVAL)
        END;


    FNSTAR:      (* CALLED FROM APPLY.  WHEN THE FUNCTION IS
                        AN F-C AND IT IS STARRED, SEARCH DEEPER
                        TO SEE IF ALL ITS MEMBERS ARE STARRED.
                        IN THAT CASE, THE STAR WILL BE DOUBLED *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("FNSTAR ");
        (* IMPORTANT... APPLY ALSO PUSHED FN AND ARGS ON THE STACK
           THEY ARE LEFT TO BE USED LATER UNDER CERTAIN CONDITIONS *)
        IF (REVAL=NIL) THEN
            PUSHONE(APPLY1,NIL)
        ELSE
            BEGIN
            LOAD(FN,ARGS);
            PUSHTWO(FN,ARGS);    (* THIS IS PROBABLY A SUPERFLUOUS ASSIGN *
            PUSHONE(FNSTAR1,ARGS)  (* THE BASE CONDITION FOR FNSTAR *)
            END
        END;


    FNSTAR1:     (* SEE FNSTAR. *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("FNSTAR1 ");
        (* FN AND ARGS ON THE STACK *)
        (* EXP IS THE ARGUMENT LIST.  IF THE F-C IS STARRED AND
            THE ARGUMENTS ALSO, PULL THE STAR OUT ONE LEVEL *)
        IF (EXP=NIL) THEN
            BEGIN
            PUSHONE(ALLSTAR,NIL);  (*CHECK FOR STARRED ARGS *)
            SETREG(REVAL,QT)
            END
        ELSE
            BEGIN
            PUSHONE(FNSTAR2,EXP);
            PUSHONE(ASTAR,NIL);
            SETREG(REVAL,EXP)
            END
        END;


    FNSTAR2:     (* CHECKING THE ARGUMENT LIST FOR ALL STARS *)
```

```
          BEGIN
          IF (TRACE>3) THEN WRITELN("FNSTAR2 ");
          (* EXP IS THE LIST WE ARE CHECKING.  FN AND ARGS ARE ON THE STAC
          IF (REVAL=NIL) THEN
              PUSHONE(FNSTAR3,EXP)
          ELSE
              PUSHONE(ALLSTAR,NIL);
          PUSHONE(ASTAR,EXP);
          PUSHONE(CAR,EXP)
          END;


      FNSTAR3:        (* SEE FNSTAR *)
          BEGIN
          IF (TRACE>3) THEN WRITELN("FNSTAR3 ");
          (* EXP IS THE LIST WE ARE CHECKING, FN AND ARGS ARE ON THE STACK
          IF (REVAL=NIL) THEN
              PUSHONE(APPLY1,NIL)
          ELSE
              BEGIN
              PUSHONE(FNSTAR4,NIL);
              PUSHONE(CDR,EXP)     (* RECUR *)
              END
          END;


      FNSTAR4:     (* SEE FNSTAR *)
          BEGIN
          IF (TRACE>3) THEN WRITELN("FNSTAR4 ");
          (* EXP IS NIL, FN AND ARGS ARE NEXT ON THE STACK *)
          PUSHONE(FNSTAR1,REVAL)
          END;


      ALLSTAR:       (* WE HAVE A STARRED F-C AND STARRED ARGUMENTS.
                              PULL THE STAR OUTSIDE OF THE APPLY *)
          BEGIN
          IF (TRACE>3) THEN WRITELN("ALLSTAR ");
          (* FN AND ARGS ARE ON THE STACK, EXP IS NIL *)
          IF (REVAL=NIL) THEN PUSHONE(APPLY1,NIL)
          ELSE
              BEGIN
              LOAD(FN,ARGS);
              PUSHTWO(FN,ARGS);
              PUSHONE(ALLSTAR1,NIL);
              PUSHONE(CARLIS,ARGS)
              END
          END;


      ALLSTAR1:        (* SEE ALLSTAR.  REVAL IS <CARLIS ARGS> *)
          BEGIN
          IF (TRACE>3) THEN WRITELN("ALLSTAR1 ");
          LOAD(FN,ARGS);
          PT:= DOTPAIR(ENVDOT,NIL);    (* BUILD A CALL TO APPLY *)
          PT:= DOTPAIR(REVAL,PT);
          PT:= DOTPAIR(MEMORY[FN].CAR.RR,PT);
          SETREG(REVAL,STAR(DOTPAIR(FAPPLY,PT),NIL))    (* STAR IT *)
```

```
        END;


    APPLY1:         (* FIRST STEP FOR AN F-C.  STRIP C'S AND CHECK
                         FOR EMPTY FUNCTIONS.  *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("APPLY1 ");
        (* EXP IS NIL, REVAL IS NOT USED *)
        LOAD(FN,ARGS);
        PT:= MEMORY[FN].CAR.RR;
        (*MOD1IF (PT=QMLIST) THEN APPLY(MEMORY[FN].CDR.RR,ARGS)
        ELSE IF (PT=QCLIST) THEN EVALERROR(3,NIL)1DOM*)
        (*MOD2*)IF (PT=QMLIST) OR (PT=QCLIST) THEN
            APPLY(MEMORY[FN].CDR.RR,ARGS)
        ELSE IF (PT=QCOLON) THEN
            EVALERROR(3,NIL)(*2DOM*)
        ELSE
            BEGIN
            PUSHTWO(FN,ARGS);
            PUSHONE(APPLY2,NIL);
            PUSHONE(ANYNULL,ARGS)
            END
        END;


    ANYNULL:     (* SEARCH AN F-C LIST FOR NIL *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("ANYNULL ");
        (* EXP IS THE LIST *)
        IF (EXP=NIL) THEN SETREG(REVAL,NIL)
        ELSE
            BEGIN
            PUSHONE(ANYNULL0,EXP);
            PUSHONE(CDR,EXP)
            END
        END;


    ANYNULL0:     (* SEARCHING A LIST FOR NIL.  SEE ANYNULL, JUST ABOVE.
                         LOOK FOR A STAR HERE, THEN GO ON. *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("ANYNULL0 ");
        (* EXP IS THE LIST, AND REVAL IS THE CDR OF EXP *)
        IF (EXP=REVAL) THEN SETREG(REVAL,NIL)
        ELSE
            BEGIN

            PUSHONE(ANYNULL1,EXP);
            PUSHONE(CAR,EXP)
            END
        END;


    ANYNULL1:     (* NOT A STAR, LOOK AT THE CAR *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("ANYNULL1 ");
        (* EXP IS THE LIST, REVAL THE CAR *)
        IF (REVAL=NIL) THEN SETREG(REVAL,QT)
```

```
        ELSE PUSHONE(ANYNULL,MEMORY[EXP].CDR.RR)
        END;


    APPLY2:   (* THE F-C LIST WAS TESTED FOR A NULL ENTRY.
                         THE RESULT OF THE TEST IS IN REVAL *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("APPLY2 ");
        (* EXP IS NIL *)
        LOAD(FN,ARGS);
        IF (REVAL=NIL) THEN
            BEGIN
            PUSHTWO(FN,ARGS);
            PUSHONE(APPLY3,NIL);
            PUSHONE(CARLIS,ARGS)
            END
        ELSE
            SETREG(REVAL,NIL)     (* A NIL IN THE F-C LIST *)
        END;


    APPLY3:      (* APPLYING AN F-C.  REVAL IS <CARLIS ARGS>.
                         EXP IS ARGS.  F-C LIST AND ARGS ON STACK *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("APPLY3 ");
        LOAD(FN,ARGS);
        PT:= DOTPAIR(ENVDOT,NIL);
        PT:= DOTPAIR(REVAL,PT);
        PT:= DOTPAIR(MEMORY[FN].CAR.RR,PT);
        PT:= DOTPAIR(FAPPLY,PT);  (* (APPLY CARFN (CARLIS ARGS) ENVDOT) *)
        PUSHTWO(PT,FN);
        PUSHONE(APPLY4,NIL);
        PUSHONE(CDRLIS,ARGS)
        END;


    APPLY4:      (* APPLYING AN F-C.  REVAL IS <CDRLIS ARGS>. *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("APPLY4 ");
        LOAD(AEXP,FN); (* AEXP=(APPLY CARFN <CARLIS ARGS> ENVDOT) *)
        PT:= DOTPAIR(ENVDOT,NIL);
        PT:= DOTPAIR(REVAL,PT);
        PT:= DOTPAIR(MEMORY[FN].CDR.RR,PT);
        PT:= DOTPAIR(FAPPLY,PT);  (* (APPLY CDRFN <CDRLIS ARGS> ENVDOT) *)
        SETREG(REVAL,CONS(AEXP,PT,NIL));
        MEMORY[REVAL].MULTI:= MEMORY[FN].MULTI
        END;


    NTH:          (* NUMERIC FUNCTION CALL.  COERCE N CDRS THEN CAR *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("NTH ");
        (* REVAL IS THE LIST.  THE PLACE INDICATOR ON TOP OF STACK IS N.
           SEE POP FOR SIDE-EFFECT ON GLOVAR PLACE *)
        POP;
        IF (REVAL=NIL) THEN BEGIN END
        ELSE IF (PLACE=1) THEN PUSHONE(CAR,REVAL)
        ELSE
```

```
003723              BEGIN
003723              PUSHONE(PLACE-1,NIL);
003727              PUSHONE(NTH,NIL);
003734              PUSHONE(CDR,REVAL)
003737              END
003741          END;
003742
003742
003742      ACAR:       (* RETURN THE CAR OF REVAL *)
003742          BEGIN
003742          IF (TRACE>3) THEN WRITELN("ACAR ");
003752          PUSHONE(CAR,REVAL)
003755          END;
003760
003760
003760      ACDR:
003760          BEGIN
003760          IF (TRACE>3) THEN WRITELN("ACDR ");
003770          PUSHONE(CDR,REVAL)
003773          END;
003776
003776
003776      ASTAR:
003776          BEGIN
003776          IF (TRACE>3) THEN WRITELN("ASTAR ");
004006          IF NOT(REVAL=NIL) THEN
004010              BEGIN
004010              PUSHONE(ASTAR1,REVAL);
004014              PUSHONE(CDR,REVAL)
004017              END
004021          END;
004022
004022
004022      ASTAR1:
004022          BEGIN
004022          IF (TRACE>3) THEN WRITELN("ASTAR1 ");
004032          (* REVAL IS THE CDR OF EXP *)
004032          IF (REVAL=EXP) THEN SETREG(REVAL,QT)
004037          ELSE SETREG(REVAL,NIL)
004044          END;
004046
004046
004046      EQ:           (* EXP IS THE ARGUMENT LIST.  THE CAR AND CADR HAVE
004046                             BEEN COERCED *)
004046          BEGIN
004046          IF (TRACE>3) THEN WRITELN("EQ ");
004056          PT:= MEMORY[EXP].CAR.RR;
004065          PT1:= CADR(EXP);
004074          IF (PT=NIL) AND (PT1=NIL) THEN SETREG(REVAL,QT)
004103          ELSE IF (NOT ISATOM(PT)) OR (NOT ISATOM(PT1))
004121              THEN SETREG(REVAL,NIL)
004124          ELSE IF MEMORY[PT].CAR.NUMBERP THEN
004133              BEGIN
004133              N:= MEMORY[PT].CAR.NN;
004136              IF MEMORY[PT1].CAR.NUMBERP AND (MEMORY[PT1].CAR.NN=N) THEN
004146                  SETREG(REVAL,QT)
004152              ELSE
004154                  SETREG(REVAL,NIL)
```

```
004160            END
004161        ELSE IF MEMORY[PT1].CAR.NUMBERP THEN SETREG(REVAL,NIL)
004172        ELSE IF (PT=PT1) THEN  SETREG(REVAL,QT)
004201        ELSE  SETREG(REVAL,NIL)
004207        END;
004211
004211
004211    ADD1:            (* REVAL IS THE ARG *)
004211        BEGIN
004211        IF (TRACE>3) THEN WRITELN("ADD1 ");
004221        IF (EXP=QATOM) THEN
004223            IF ISATOM(REVAL) THEN SETREG(REVAL,QT)
004233            ELSE SETREG(REVAL,NIL)
004240        ELSE IF (EXP=QNULL) OR (EXP=QNOT) THEN
004246            BEGIN
004246            IF (REVAL=NIL) THEN SETREG(REVAL,QT)
004253            ELSE SETREG(REVAL,NIL)
004260            END
004261        ELSE IF (REVAL=NIL) THEN EVALERROR(4,REVAL)
004267        ELSE IF NOT MEMORY[REVAL].CAR.NUMBERP THEN EVALERROR(4,REVAL)
004301        ELSE
004303            BEGIN
004303            N:= MEMORY[REVAL].CAR.NN;
004307            IF (EXP=QADD1) THEN N:= N+1
004310            ELSE N:= N-1;
004312            SETREG(REVAL,MAKENUM(N))
004324            END
004325        END;
004326
004326
004326    PLUS:       (* THE ARGS HAVE BEEN COERCED, AS IN EQ.  THE CALL
004326                         TO PLUS IS IN PROCEDURE APPLY.  SELECT
004326                         THE PROPER BINARY OPERATION *)
004326        BEGIN
004326        IF (TRACE>3) THEN WRITELN("PLUS ");
004336        (* ON THE STACK: FUNCTION-NAME, ARGUMENT LIST *)
004336        LOAD(FN,ARGS);
004341        (*REVAL IS THE CAR OF ARGS *)
004341        PT:= CADR(ARGS);
004350        IF (NOT MEMORY[REVAL].CAR.NUMBERP) THEN EVALERROR(4,REVAL)
004357        ELSE IF (NOT MEMORY[PT].CAR.NUMBERP) THEN
004366            BEGIN
004366            SETREG(REVAL,PT);
004372            EVALERROR(4,PT)
004375            END
004377        ELSE
004400            BEGIN
004400            N:= MEMORY[PT].CAR.NN;
004404            I:= MEMORY[REVAL].CAR.NN;
004407            IF (FN=QGREAT) THEN
004411                BEGIN
004411                IF (I>N) THEN SETREG(REVAL,QT)
004415                ELSE SETREG(REVAL,NIL)
004422                END
004423            ELSE IF (FN=QLESS) THEN
004425                BEGIN
004425                IF (I<N) THEN SETREG(REVAL,QT)
004432                ELSE SETREG(REVAL,NIL)
```

```
                END
            ELSE
                BEGIN
                IF (FN=QPLUS) THEN N:= N+I
                ELSE IF (FN=QDIFF) THEN N:= I-N
                ELSE IF (FN=QTIMES) THEN N:= I*N
                ELSE IF (FN=QDIV) THEN N:= I DIV N
                ELSE IF (FN=QMOD) THEN N:= N MOD I;
                SETREG(REVAL,MAKENUM(N))
                END
            END
        END;


CARLIS:     (* RETURNS THE FIRST ARGUMENT FOR AN F-C
                    EXP IS THE ARGUMENT ARRAY *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("CARLIS ");
        IF (EXP=NIL) THEN SETREG(REVAL,NIL)
        ELSE
            BEGIN
            PUSHONE(CARLIS1,EXP);
            PUSHONE(CDR,EXP)
            END
        END;


CARLIS1:      (* REVAL IS THE CDR OF EXP.  CHECK FOR A STAR
                    THEN GET THE CAAR OF THE LIST.   *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("CARLIS1 ");
        IF (REVAL=EXP) THEN
            BEGIN
            PT:= DOTPAIR(QEXP,NIL);
            PT:= DOTPAIR(QCAR,PT);
            PT:= DOTPAIR(PT,NIL);
            PT:= DOTPAIR(QCAR,PT);    (* MAKING (CAR (CAR (QUOTE EXP))) *)
            PT1:= DOTPAIR(QEXP,EXP);
            PT1:= DOTPAIR(PT1,NIL);
            PT2:= DOTPAIR(NIL,NIL);
            PT2:= DOTPAIR(PT1,PT2);      (* MAKING AN ENVIRONMENT *)
            SETREG(REVAL,STAR(PT,PT2))   (* SUSPENDED, STARRED CALL *)
            END   (* END THE STARRED CASE *)
        ELSE
            BEGIN
            PUSHONE(CARLIS2,EXP);
            PUSHONE(ACAR,NIL);
            PUSHONE(CAR,EXP)
            END
        END;


CARLIS2:        (* THE ARGS ARE IN EXP AND THEY ARE NOT
                    STARRED.  REVAL HAS THE CAAR OF ARGS *)
        BEGIN
        IF (TRACE>3) THEN WRITELN("CARLIS2 ");
        IF (REVAL=QHSH) THEN PUSHONE(CARLIS,MEMORY[EXP].CDR.RR)
        ELSE
```

```
004745            BEGIN
004745            PT:= DOTPAIR(QCARLIS,MEMORY[EXP].CDR.RR);
004763            PT:= CONS(NIL,PT,NIL);
004776            MEMORY[PT].CAR.RR:= REVAL;
005005            REVAL:= PT;              (* STEAL REVAL'S REFERENCE FOR CON
005007            NUDGE(REVAL)
005011            END
005012        END;
005013
005013
005013    CDRLIS:          (* WORKS LIKE CARLIS JUST ABOVE *)
005013        BEGIN
005013        IF (TRACE>3) THEN WRITELN("CDRLIS ");
005023        IF (EXP=NIL) THEN SETREG(REVAL,NIL)
005030        ELSE
005032            BEGIN
005032            PUSHONE(CDRLIS1,EXP);
005036            PUSHONE(CDR,EXP)
005041            END
005043        END;
005044
005044
005044    CDRLIS1:          (* CHECK FOR A STAR, THEN GO ON *)
005044        BEGIN
005044        IF (TRACE>3) THEN WRITELN("CDRLIS1 ");
005054        PT:= DOTPAIR(FCDAR,EXP);
005066        IF (EXP=REVAL) THEN SETREG(REVAL,STAR(PT,NIL))
005103        ELSE
005105            BEGIN
005105            PT1:= DOTPAIR(QCDRLIS,MEMORY[EXP].CDR.RR);
005123            SETREG(REVAL,CONS(PT,PT1,NIL))
005141            END
005142        END;
005143
005143
005143    EVLIS:        (* THIS IS A TYPICAL RECURSION.
005143                        START HERE TO UNDERSTAND HOW MOST
005143                        OF THESE CASES WORK.  CHECK FOR THE
005143                        BASE CONDITION HERE, THEN COERCE THE
005143                        CDR AND RECUR *)
005143        BEGIN
005143        IF (TRACE>3) THEN WRITELN("EVLIS ");
005153        (* THE POP WHICH FOUND THIS CASE ALSO LOADED THE
005153           REGISTER "EXP".  ITS VALUE HERE IS THE LIST *)
005153        IF (EXP=NIL) THEN SETREG(REVAL,NIL)
005160        ELSE
005162            BEGIN
005162            PUSHONE(EVLIS1,EXP);
005166            PUSHONE(CDR,EXP)
005171            END
005173        END;
005174
005174
005174    EVLIS1:        (* DO THE SUSPENDED CONS.  REVAL IS THE
005174                        COERCED CDR OF THE LIST.  CHECK
005174                        FOR A STAR CONFIGURATION, THEN DO THE
005174                        CONS.   *)
005174        BEGIN
```

```
          IF (TRACE>3) THEN WRITELN("EVLIS1 ");
          PT:= DOTPAIR(QEVCAR,EXP);
          IF (EXP=REVAL) THEN SETREG(REVAL,STAR(PT,ENVDOT))
          ELSE
              BEGIN
              PT1:= DOTPAIR(QMLIST,REVAL);
              SETREG(REVAL,CONS(PT,PT1,ENVDOT))
              END;
          MEMORY[REVAL].MULTI:= MEMORY[EXP].MULTI
          END;


      END;    (* END OF THE CASE STATEMENT *)


IF (STAQUE=NIL) THEN
      BEGIN
      IF (STACK=NIL) THEN ALLDONE:= TRUE
      ELSE IF (MODECODE=0) THEN
          BEGIN
          SETREG(REVAL,STACK);
          ALLDONE:= TRUE
          END
      END
ELSE
      BEGIN
      IF (STACK=NIL) THEN CONTEXTPOP(REVAL);
      WHILE (MODECODE=0) AND (NOT (STAQUE = NIL)) DO CONTEXTPOP(STACK);
      IF (STAQUE=NIL) THEN
          BEGIN
          SETREG(REVAL,STACK);
          ALLDONE:= TRUE
          END
      END;
IF NOT ALLDONE THEN
      BEGIN
      IF (TRACE>5) THEN WRITELN;
      POP;
      IF (TRACE>5) THEN WRITE("-=>")
      END



END  (* END OF THE WHILE STATEMENT *)
END; (* END OF PROCEDURE EVAL *)






PROCEDURE READLOOP(INDEVICE,OUTDEVICE:INTEGER);
                        (* A TRANSCIENT VERSION OF THE READ-EVALUATE-
                           WRITE PROCESS *)
      LABEL 1,2,3,4,5,6;
      VAR P,Q,STACK: PTR;
          (* P AND Q ARE PROCESS REGISTERS.
             STACK INVARIANT: NUDGE EACH NODE WHEN IT IS PUT ON THE
```

```
000010                          STACK, AND REDUCE ITS REF-CNT WHEN IT
000010                          IS REMOVED.  THE VARIABLE 'STACK' IS
000010                          TREATED LIKE A LOCAL REGISTER (NO SETREGS)
000010                          TO AVOID SOME NUDGES *)
000010     BEGIN
000010     P:= NIL; Q:= NIL; STACK:= NIL;
000011     WHILE TRUE DO (* READ LOOP *)
000012         BEGIN
000012         WRITELN;
000013         OUTPOINT:= OUTPUTSIZE;
000014         Q:= MYREAD;
000020         IF (TRACE>1) THEN
000021             BEGIN
000021             WRITELN;
000022             WRITE("-=>-=>-=> ENTER READ LOOP.  Q IS ",Q:1);
000033             WRITENODE(Q,TRUE)
000040             END;
000041         IF (Q=QSTOP) THEN GOTO 5;
000044         EVAL(PROCESS(TOP,Q),37);
000060         WHILE SUSPENDED(REVAL) DO
000065             BEGIN
000065             Q:= REVAL; REVAL:= NIL;
000071             EVAL(Q,37)
000073             END;
000075         Q:= NIL; SETREG(Q,REVAL); SETREG(REVAL,NIL);
000106         PUTCH(PRINC,"-"); PUTCH(PRINC,"="); PUTCH(PRINC,">");
000130         IF ISATOM(Q) THEN
000135             BEGIN
000135             IF (Q=NIL) THEN
000136                 BEGIN
000136                 PUTCH(PRINC,"(");
000144                 PUTCH(PRINC,")")
000151                 END
000152             ELSE PUTAT(Q);
000157             RECYCLE(Q);
000163             GOTO 4
000164             END;
000164         PUTCH(PRINC,BLANK); PUTCH(PRINC,"(");
000200         WHILE TRUE DO   (* PRINT LOOP *)
000201         BEGIN
000201         WHILE TRUE DO   (* CAR LOOP *)
000202             BEGIN
000202             SETREG(P,Q); SETREG(Q,NIL);
000212             EVAL(PROCESS(CAR,P),37);
000226             WHILE SUSPENDED(REVAL) DO
000233                 BEGIN
000233                 Q:= REVAL; REVAL:=NIL;
000237                 EVAL(Q,37)
000241                 END;
000243             Q:= NIL; SETREG(Q,REVAL); SETREG(REVAL,NIL);
000254             SETREG(EXP,NIL);
000260             IF (TRACE>1) THEN
000262                 BEGIN
000262                 WRITELN;
000263                 WRITE("-=>-=>-=> IN CAR LOOP. CAR OF P IS ",Q:1);
000274                 WRITENODE(Q,TRUE)
000301                 END;
000302             IF (Q=NIL) THEN
```

```
000304                BEGIN
000304                PUTCH(PRINC,"("); PUTCH(PRINC,")");
000320                GOTO 1
000321                END;
000321            IF (Q=P) THEN
000323                BEGIN
000323                PUTCH(PRINC,"*");
000331                GOTO 1
000332                END;
000332            IF ISATOM(Q) THEN
000337                BEGIN
000337                PUTAT(Q);
000343                GOTO 1
000344                END;
000344            PUTCH(PRINC,"(");
000352            MEMORY[P].CAR.RR:= STACK;
000363            STACK:= P; NUDGE(P);
000367            END; (* CAR LOOP *)
000370 1:     WHILE TRUE DO   (* CDR LOOP *)
000371            BEGIN
000371            SETREG(Q,NIL);
000375            EVAL(PROCESS(CDR,P),37);
000411            WHILE SUSPENDED(REVAL) DO
000416                BEGIN
000416                Q:= REVAL; REVAL:= NIL;
000422                EVAL(Q,37)
000424                END;
000426            Q:= NIL; SETREG(Q,REVAL); SETREG(REVAL,NIL);
000437            SETREG(EXP,NIL);
000443            IF (TRACE>1) THEN
000445                BEGIN
000445                WRITELN;
000446                WRITE("-=>-=>-=> IN CDR LOOP.   CDR OF P IS ",Q:1);
000457                WRITENODE(Q,TRUE)
000464                END;
000465            IF (Q=NIL) THEN SETREG(Q,P)
000472            ELSE
000474                BEGIN
000474                IF (Q=P) THEN
000475                    BEGIN
000475                    PUTCH(PRINC," "); PUTCH(PRINC,"*");
000511                    GOTO 6
000512                    END
000512                ELSE IF (MEMORY[P].REF.NN <> 1) THEN
000517                    BEGIN
000517                    MEMORY[P].CDR.RR:= STACK;
000525                    STACK:= P; NUDGE(P);
000532                    MEMORY[P].PNAME:= TRUE
000535                    END;
000540                PUTCH(PRINC," ");
000546                IF ISATOM(Q) THEN
000553                    BEGIN
000553                    PUTCH(PRINC,"."); PUTCH(PRINC," ");
000566                    PUTAT(Q)
000571                    END
000572                ELSE
000573                    GOTO 3
000574                END;
```

```
6:              PUTCH(PRINC,")");
                WHILE TRUE DO    (* POP LOOP *)
                    BEGIN
                    IF (STACK = NIL) THEN GOTO 4;
                    MEMORY[STACK].REF.NN:= MEMORY[STACK].REF.NN-1;
                    SETREG(P,STACK);
                    IF (NOT MEMORY[P].PNAME) THEN GOTO 2;
                    MEMORY[P].PNAME:= FALSE;
                    STACK:= MEMORY[P].CDR.RR;
                    MEMORY[P].CDR.RR:= Q;
                    SETREG(Q,P)
                    END;    (* POP LOOP *)
2:              STACK:= MEMORY[P].CAR.RR;
                MEMORY[P].CAR.RR:= Q;
                END;    (* CDR LOOP *)
3:          END;    (* PRINT LOOP *)
4:          END;    (* READ LOOP *);
5:  SETREG(Q,NIL); SETREG(P,NIL)
    END;    (* PROCEDURE READLOOP *)



(*+++++++ MAIN PROCEDURE +++++++*)

BEGIN
   LINELIMIT(OUTPUT,-1);
   (* INITIALIZE PATTERN NODES *)
   WITH READPUSH DO
   BEGIN
      ATOMP:= FALSE;
      MULTI:= FALSE;
      PNAME:= FALSE;
      REF.NUMBERP:= FALSE;
      CAR.NUMBERP:= FALSE;
      CDR.NUMBERP:= FALSE;
           CAR.RR:=NIL;
           CDR.RR:=NIL;
           REF.RR:=NIL;
   END;
   WITH NEWSUSPEND DO
   BEGIN
      ATOMP:= FALSE;
      MULTI:= FALSE;
      PNAME:= FALSE;
      REF.NUMBERP:= FALSE;
      CAR.NUMBERP:= FALSE;
      CDR.NUMBERP:= FALSE;
      REF.RR:= NIL;
      CAR.RR:=NIL;
      CDR.RR:= NIL;
   END;
   STACKPUSH:= NEWSUSPEND;
      STACKPUSH.CAR.NUMBERP:= TRUE;
      STACKPUSH.CAR.NN:= 0;
   WITH NEWCONS DO
   BEGIN
      ATOMP:= FALSE;
      MULTI:= FALSE;
```

```
000700          QARGS:=SYSATM("ARGS     ",3);
000705          QFUNCTIO:=SYSATM("FUNCTION",7);
000712          QFUNARG:=SYSATM("FUNARG  ",5);
000717          QLAMBDA:=SYSATM("LAMBDA  ",5);
000724          QREDEF:=SYSATM("REDEF   ",4);
000731          QSQB:=SYSATM("SQB     ",2);
000736          QCOLON:=SYSATM("##:##   ",4);
000743          QAP:=SYSATM("AP      ",1);
000750          QFP:=SYSATM("FP      ",1);
000755          QSTARRED:=SYSATM("STARRED ",6);
000762          LIST:=FALSE;
000763          CALLRECLAIM:= 0;
000764          POINT:= DOTPAIR(QT,NIL);
000775          ALIST:= DOTPAIR(QT,NIL);
001006          ALIST:= DOTPAIR(ALIST,POINT);
001020          NUDGE(ALIST);
001023          ENVDOT:= NIL;
001024          READACHAR:= TRUE; CARRAIGERETURN:= FALSE;
001025          SPEAK:=0;
001026          OUTPOINT:= OUTPUTSIZE;
001027          FOR I:=1 TO OUTPUTSIZE DO
001030           OUTIMAGE[I]:=BLANK;
001037          THISCH:=BLANK;
001040          POINT:=DOTPAIR(QLIS,NIL);
001052          POINT:=DOTPAIR(QLIS,POINT);
001064          POINT:=DOTPAIR(QLAMBDA,POINT);
001076          POINT:= DOTPAIR(POINT,NIL);
001107          FLIST:= DOTPAIR(QMLIST,NIL);
001120          FLIST:= DOTPAIR(FLIST,POINT);
001132          NUDGE(FLIST); NUDGE(FLIST);
001141          FINIS:=FALSE;
001142          CALLRECLAIM:=0;RETURNS:=0;DRETURNS:=0;
001144          WRITELN(" ---=>--=>-=> ENTERING VERSION 0.1");
001152          WRITELN;
001153          DORECLAIM:= TRUE;
001154          WRITELN("-=> MEMORY LIMIT);
001162          READ(MEMORYLIMIT);
001165          MEMORY[MEMORYLIMIT].CDR.RR:= NIL;
001173          READLOOP(0,0);
001174 (*++++++++++++++++++DEBUGGING HERE+++++++++++++++++++++++++*)
001174 WRITELN; WRITELN("-=>-=>-=>-=> LEAVING.");
001203    WRITELN("NODES DISPOZED, ",DRETURNS:1);
001215    WRITELN("NODES RECYCLED, ",RETURNS:1);
001227    WRITELN("AVAIL--> ",AVAIL:1);
001241
001241
001241
001241
001241
001241 END.
```

Curriculum Vitae

Steven Dexter Johnson is currently with Bell Laboratories in Holmdel, New Jersey. He was born in Billings, Montana in 1948. Mr. Johnson received his Bachelor's Degree in Mathematics and Russian from DePauw University in 1970. He received a Master of Arts Degree in Mathematics from Indiana University in 1972. He has been a member of the Association for Computing Machinery since 1975.