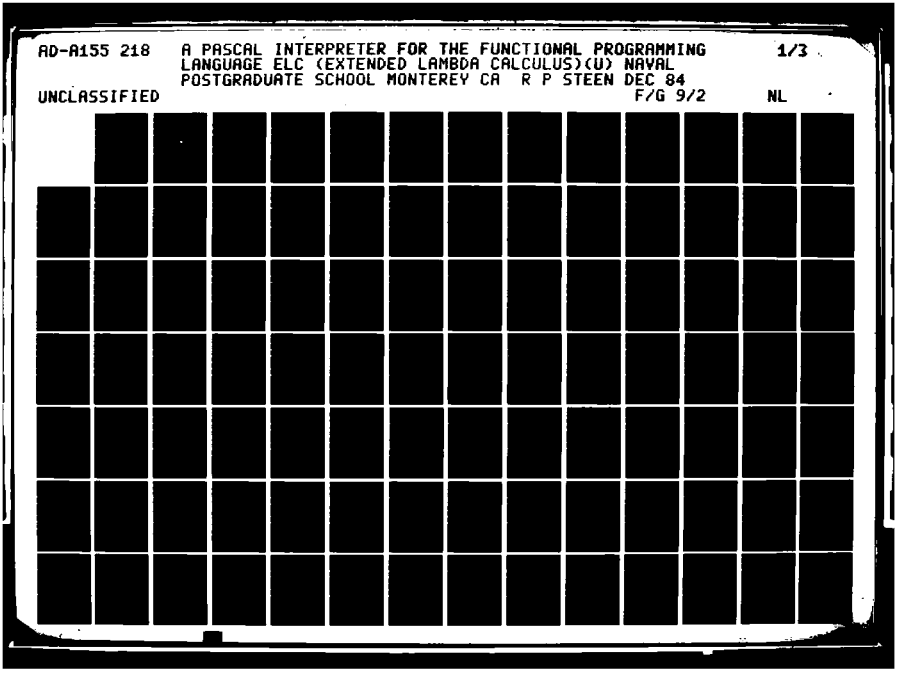
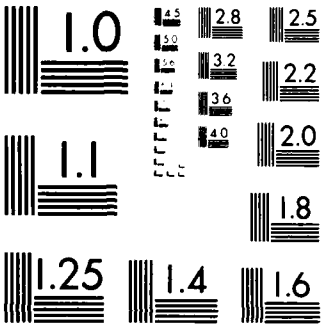
AD-A155 218 A PASCAL INTERPRETER FOR THE FUNCTIONAL PROGRAMMING LANGUAGE ELC (EXTENDED LAMBDA CALCULUS)(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA R P STEEN DEC 84 1/3

UNCLASSIFIED F/G 9/2 NL





NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A PASCAL INTERPRETER FOR THE FUNCTIONAL PROGRAMMING LANGUAGE ELC

by

Ralph P. Steen, Jr.
December 1984

Thesis Advisor: Bruce J. MacLennan

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A155 218
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) A Pascal Interpreter for the Functional Programming Language EQL	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December 1984
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Ralph P. Steen, Jr.	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93493	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943	12. REPORT DATE December 1984
	13. NUMBER OF PAGES 255
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

functional programming, abstraction, closure, recursion, environment, reference counts.

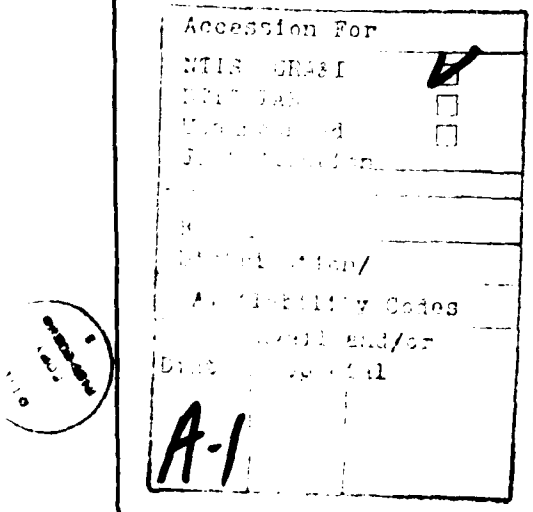
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Functional programming is a methodology designed to eliminate many of the problems of past programming languages through actions such as the elimination of the assignment statement and the ability to program in an environment that is at a higher level of abstraction than any previous languages. In this report an interpreter, written in Pascal, for the Extended Lambda Calculus is presented. Initially, the

(Continued)

ABSTRACT (Continued)

events leading to the development of functional programming is discussed followed by an in depth look at how the interpreter operates. Numerous example ELC programs are presented, including discussions of practical applications and statistical information about execution times and memory requirements. The Berkeley Pascal source code for the interpreter is also included in Appendix C.



Approved for public release; distribution is unlimited.

A Pascal Interpreter for the Functional Programming Language ELC

by

Ralph P. Steen Jr.
Captain, United States Army
B.S., United States Military Academy, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1984

Author: Ralph P. Steen Jr.
Ralph P. Steen Jr.

Approved by: B.J. MacLennan
B.J. MacLennan, Thesis Advisor

Gordon Bradley
Gordon Bradley, Second Reader

B.J. MacLennan
B.J. MacLennan, Chairman,
Department of Computer Science

Kneale T. Marshall
Kneale T. Marshall
Dean of Information and Policy Sciences

ABSTRACT

Functional programming is a methodology designed to eliminate many of the problems of past programming lanuages through actions such as the elimination of the assignment statement and the ability to program in an environment that is at a higher level of abstraction than any previous languages. In this report an interpreter, written in Pascal, for the Extended Lambda Calculus is presented. Initially, the events leading to the development of functional programming is discussed followed by an in depth look at how the interpreter operates. Numerous example ELC programs are presented, including discussions of practical applications and statistical information about execution times and memory requirements. The Berkeley Pascal source code for the interpreter is also included in Appendix C.

TABLE OF CONTENTS

LIST OF FIGURES

I. PURPOSE AND BACKGROUND

A. PURPOSE

The purpose of this thesis it to illustrate the design and use of an interpreter for the Extended Lambda Calculus (ELC) as described by MacLennan [Ref. 1]. Initially, however, it is important to understand why functional languages such as ELC are important and why they will become increasingly important in the future. To achieve this, a brief background sketch is presented to explain the events that have shaped the need for such languages.

B. BACKGROUND

During the brief history of Computer Science there has been a remarkably rapid evolution of computing hardware, while software development has for all practical purposes remained static. Throughout the last thirty years, improvements such as: decreasing component size, increased memory capacity, faster processor speeds and reduced hardware costs have occurred at regular intervals. This trend continues today in areas such as super computers like the Cray and Cyber and the rapidly changing micro computer industry. If one studies the evolution of computer software for the same time period, in particular programming languages, the same types of rapid improvements on a regular basis have not occurred. The first revolutionary development in programming languages occurred with the development of FORTRAN by Backus et al. in the mid 1950s. For the first time scientific programmers could write code that strongly resembled the equations they were working with. Practically all the programming languages developed since that time, perhaps

with the exception of LISP and APL, are basically the same underneath as Fortran. Of course there are some outward differences, such as sophisticated string and array handling mechanisms, but they are all sequentially processed and rely heavily on use of the assignment statement, variables and the notion of machine state. The early pioneers in programming languages are not totally at fault for the lack of progress. To understand this statement, a brief examination of the architecture these languages were written for is necessary.

The great improvements in hardware development, mentioned previously, were also not fundamental until fairly recently. Hardware improvements remained superficial in that the majority of them were made on the same architecture, that proposed by von Neumann et al. in 1946. Briefly, the von Neumann architecture consists of a Central Processing Unit, a Memory used to store both programs and data, and some kind of connection between the two capable of transmitting single words or addresses back and forth. Improvements have concentrated on decreasing size, increasing speed, etc. and have not been concerned about the basic design of the computer. The connection between the memory and the CPU is the reason why most programming languages are sequential in nature, forcing the user to deal with some fairly low level constructs such as incrementing counters and setting up iteration loops. Backus [Ref. 2] termed this connection the von Neumann bottleneck and also described conventional programming languages as just software versions of von Neumann machines. Computer architectures are starting to change, however, and in order to gain the maximum benefit from them programming languages and techniques must change also.

Throughout the development of new hardware systems the trend has been to increase speed by making components

smaller and smaller. Common sense dictates that eventually the ability to do this will become physically impossible. Does this mean that the quest to increase computation speed will stop? Obviously not. The most promising solution is to fully exploit parallel operations in data processing whenever possible. As explained by Stone [Ref. 3], much promising work has been accomplished in the fields of array, multiprocessor, and pipeline computers, but there are still open research problems concerning how to properly organize and synchronize all these processors. In other words there is no effective software to manage parallel computer operations. Conventional progamming languages, with their sequential nature, do not offer much hope as a solution. The functional programming languages, such as FP proposed by Backus [Ref. 2], or the Kent Recursive Calculater by Turner [Ref. 4], by their very nature lend themselves to parallel operations. This is illustrated in the next section.

As stated previously, one of the biggest problems with conventional languages is the assignment statement. Backus [Ref. 2] calls the assignment statement the bottleneck of programming languages because at the heart of all conventional programs we find a myriad of assignment operations producing one word results. The programmer must then concern himself with the flow of words through these assignment statements to achieve the desired results, instead of concerning himself with the problem as a whole. Another problem with the assignment statement is that it makes programs unreliable. Mathematical proofs do not lend themselves well to statements. Consider someone trying to do a mathematical proof of the following statement.

x := x+1

That statement makes absolutely no sense mathematically. How can 'x' be assigned the value of itself plus one? This statement is legal, however, in most conventional

programming languages and makes formal proofs of them extremely difficult as shown by the work of Hoare [**Ref. 5**]. Functional languages do away with the idea of the assignment statement and work only with expressions. Expressions, in contrast with statements, do posess mathematical properties. Backus [**Ref. 2**] has even shown that the functional language FP lends itself to an algebra of programs that can allow for relativly simple proofs of program correctness. Since functional languages deal only with expressions, the idea of execution order becomes obsolete, as explained by MacLennan [**Ref. 6**]. One begins to understand how these languages can be used to exploit parallelism since several expressions could be solved simultaneously and then brought together to form a final result.

Another advantage of functional programming is the compactness of the code written by programmers. Two examples from the literature illustrate this fact very well. Backus shows in [**Ref. 2**] an FP program to calculate the factorial of an arbitrary integer n. The program is one line long, whereas the corresponding program written in PL/I is eight lines long. The two programs are shown in Figure 1.1 for comparison. An even more startling example was devised by Early [**Ref. 7**], where a two pass assembler was written in both FP and C for an artificial assembly language. The assembler written in C occupied 459 lines of non-comment source code, where as the FP assembler occupied 249 lines, of which more than 100 lines were only a single char[illegible] so as to aid in program readability and clarity. This fact could have far reaching effects in an attempt to solve the software crisis as presented by Turner [**Ref. 4**]. The fact that the code is more compact could mean increased programmer productivity since it is well known that programming time is roughly proportional to the number of lines of code regardless of the language being used. Also, since it

memb <2> <1 2 3> = false
memb <a b> <z t s <a b>> = true

E. THE HEART OF THE INTERPRETER, THE EVAL FUNCTION

The eval function, Figure 2.6, is the most important function in the interpreter. Eval acts as a decoder, determining how each list sent to it is to be interpreted. This is accomplished by stripping off the first element of the list or program and then invoking a rule that corresponds to that first element. For example, if the program is

<list a b c>

eval will strip off the word "list" and return <a b c> as the result. Referring back to the line of code that starts program execution, it is seen that after the program is read in it is sent to the eval function along with a pointer called primitives. Primitives is a pointer to an association list which acts as an environment for executing the primitive operations discussed in the last section. For a detailed discussion of association lists and environments see MacLennan [Ref. 6]. The primitives association list is built initially by using the dcprim (declare primitives) function. An example of a part of the list is shown in Figure 2.7. The reason this primitives association list is constructed is to maintain consistency between how primitive and user defined functions are evaluated by the interpreter. This is discussed in more detail later in the next section.

It is important to now look at the key words recognized by eval and the rules they invoke. By doing this, a complete understanding of the interpreter will be achieved.

1. Function Eval's Keywords

- list 'List' simply lets the interpreter know that this expression is a list, so eval returns the rest of the list sent to it. For example, if eval is sent

- Sub Returns a particular element from a list. Takes two arguments: a pointer to a list and an integer indicating the position of the desired element in a list. For example, sub (<A B C> 2) = B

- Repr Takes a finite set (finset) as an argument and returns its ELC representation, which is simply a list.
  repr <finset a b c 1 2> = <a b c 1 2>

- Len Determines the length of any list.
  len <a> = 1
  len < > = 0
  len <a <b c> d> = 3

- Equal Equal tests the equality of any two atoms or any two lists.
  equal <2 2> = true
  equal <a c> = false
  equal <a b c> <a b c> = true
  equal <a b c> <a <b> c> = false

- GT Greater-than tests if arg1 is greater that arg2.
  GT arg1 arg2 = true/false
  GT 2 3 = false
  GT 5 1 = true

The next three boolean primitives follow the same pattern as GT.

- LT Less-than

- GE Greater-than or equal to

- LE Less-than or equal to

- Memb Member tests to see if arg1 is an element of arg2, which must be a list.
  memb arg1 <arg2> = true/false
  memb 2 <1 2 3> = true

conr a <b c> = <b c a>
conr <a b> <c d> = <c d <a b>>

- Atom A Boolean function that determines if its argument is an atom.

  atom 'a' = true
  atom <list a b c> = false

- Null A Boolean function to determine if a list contains no elements. The last example is a list containing one element which happens to be a null list.

  null <> = true
  null <a> = false
  null <<>> = false

- Binary Arithmetic Operators Each of the listed operators works for any m,n where m,n are two integers or two real numbers.

  sum m n
  subt m n
  prod m n
  divi m n

- Trigonometric Functions The following functions take single arguments of angles in degrees.

  sin x
  cos x
  tan x
  cot x
  sec x
  csc x

- Id Identity Function; simply returns the argument it is sent, eg. id 2 = 2 and id 'a' = 'a'. The purpose of this function is illustrated in example program x Appendix B which generates the table of sin, cos, tan for all angles from 0 to 90 degrees.

description of the primitive cons is covered in Section 3 o this chapter, but suffice it to say that cons forms the first element of the list represented as a cell pointed to by the head of a lst cell with the tail set to nil. Additional cells are added by connecting them properly to the tail of the last cell read using the cons primitive and manual manipulation of pointers. This is accomplished by the while loop in function readlist. This process continues until a right angle bracket is recognized, which of course signifies the end of a list.

3. Primitive Operations

The following are the primitive operations provided by the interpreter and a brief explanation of each. Correct syntax for the language is covered in Appendix A.

- First Takes a list and returns the first element, eg. the first of <a b c> is a.
- Rest Takes a list and returns a list containing everything but the first element, e.g., the rest of <a <b c d> e> is <<b c d> e>.
- Last Takes a list and returns the last element, e.g., the last of <a b c> is c.
- Initial Takes a list and returns all elements except the last, e.g., the initial of <a b c> is <a b>.
- Cons Takes an atom or list and makes it the first element of a second list. The second argument of a cons operation must be a list.

  cons a <b c> = <a b c>

  cons <a b> <c d> = <<a b> c d>

- Conr Cons to the right. Conr is the opposite of the cons operation in that it makes an atom or list the last element of a second list.

number of lst cells that are at the top of the Figure, there are three with the tail field of the lst cell at the far right set to nil. Being consistent with the previous discussion this is a list containing three elements. The head of the second lst cell, however, does not point to a leaf or information cell, it points to another lst cell which forms the identical structure as previously seen. Again, the tail field of the last element in the internal list is nil, signifying the end of this list.

D. ELC PROGRAMS

1. Definition

An ELC program is nothing more than a list built in such a way that it can be evaluated by the interpreter. It is important to remember that one program equals one list. Of course this one list usually consists of many other nested lists as its elements.

2. Reading Programs

Programs are read into the interpreter by the readval and readlist functions, which can be reviewed in Appendix C, Source Code. The reading process is started by the following line of the interpreter.

```
printval ( eval( readval, primitives))
```

The first function called is the readval function which determines the type of data being read by recognizing the first character of the input. Since a program is a list, the first character will obviously be a left angle bracket '<', transfering execution to the readlist function. Readlist builds the program into the same kind of structure as discussed in the last section. This is seen by first noticing that readlist calls readval again and the results are placed in a list through the cons function. A detailed

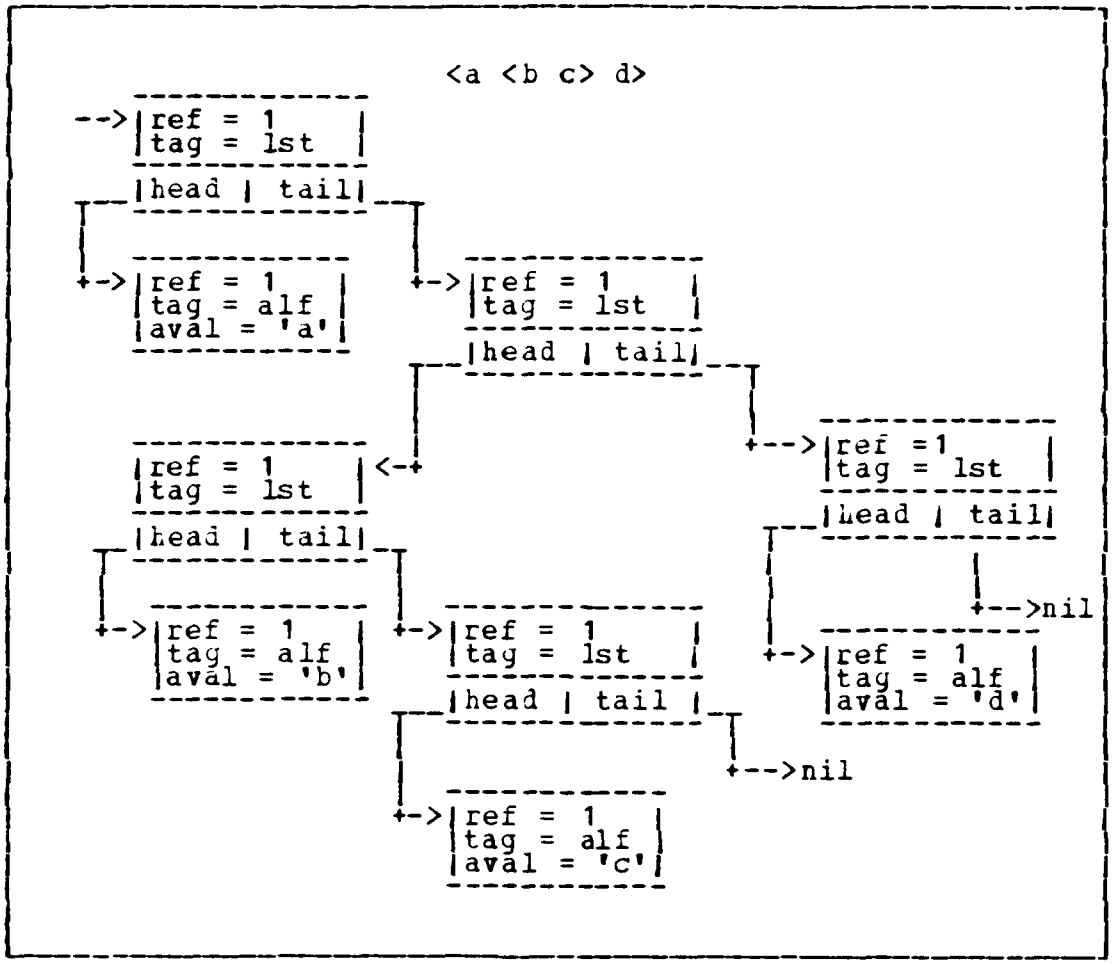


Figure 2.5 List Within a List.

The first list contains three atoms. It is easy to tell that there are three elements in the list by counting the cells that are tagged as lists (lst). The information in the list is contained in other cells that are pointed to by the heads of the list cells. The tails of the list cells point to the next list cell, which in turn points to the next element in the list. The end of the list is represented by the tail field of the last element being set to nil. The example in Figure 2.5 is a list that has as one of its elements another list. Once again if you count the

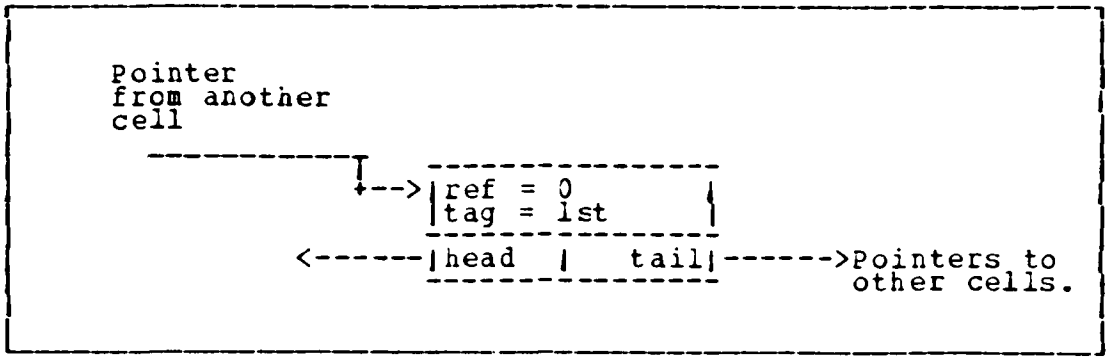


Figure 2.3 Pascal Representation of a List Cell.

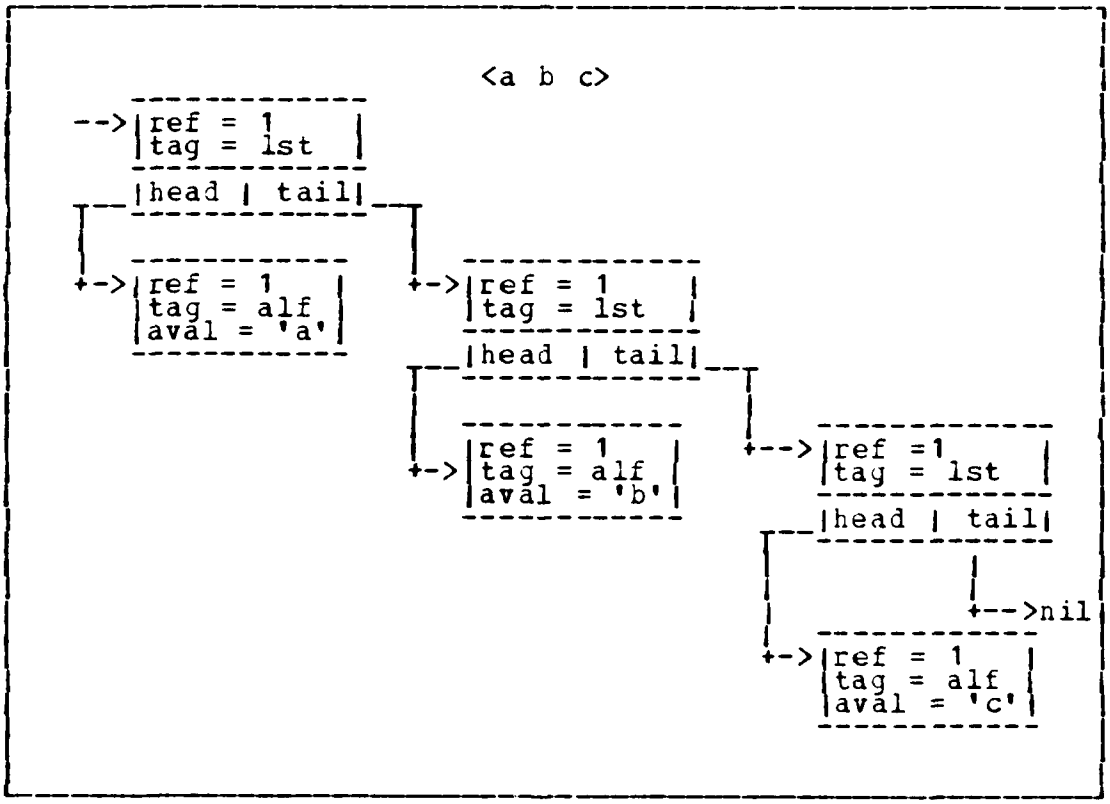


Figure 2.4 Representation of a Simple List.

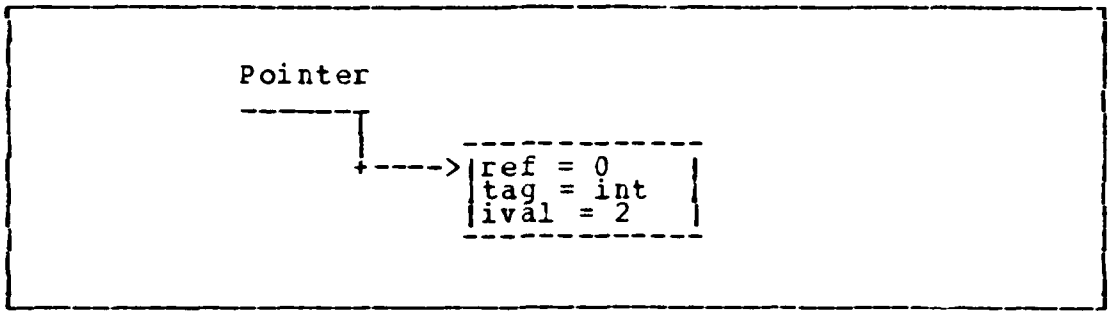


Figure 2.2 Representation of a Cell Containing an Integer.

2. What about lists?

The atoms are the building blocks of ELC and, when placed in sequences form, the lists previously described. Lists are also formed using variant records. Lists are naturally thought of as items that are grouped together because of a common bond. The linked list of Pascal is the natural method to use to represent these groups. This is clear because the interpreter needs to be able to create lists of varying lengths during execution. Since the size of Pascal arrays must be declared before program execution, their use to represent lists is impossible. A language with arrays that could grow dynamically would be more efficient to use in order to avoid the overhead required in maintaining Pascal pointers in linked lists. The list cell is shown in Figure 2.3. The basic cell structure is the same, except that the tag is now 'lst' and the variant portion of the record contains two fields (head and tail) that are pointers to other cells. As mentioned previously, lists are sequences of atoms, lists, or atoms and lists surrounded by angle brackets. In order for the interpreter to recognize where the angle brackets are, the tail field of certain lst cells are set to nil. The representation of two simple lists is shown in Figures 2.4 and 2.5.

refer to the grammar in Appendix A. Lists are sequences of atoms or lists or atoms and lists separated by spaces and surrounded by angle brackets as in Figure 2.1.

```
<list a b c>
<list a b <list c d> e>
<letrec append . . .>
```

Figure 2.1 ELC Lists.

Atoms are used to represent information and data in the language, so the interpreter must have a way of representing them. Simple records are not adequate, however, because there are several kinds of atoms and they must be distinguishable. The perfect choice is the variant record, which allows the same record structure to be used for all atoms while permitting some or all of the information to vary depending on the value of a tag field. These variant records are referred to as cells throughout the remainder of the report. Tags for the different atoms are:

- boo (boolean values)
- rea (real values)
- int (integer values)
- alf (identifier; corresponds to Berkeley Pascal alfa type which is a string of ten characters)

Figure 2.2 illustrates the Pascal represention of an atom cell.

Memory management is covered in detail in Section F of this chapter. The other advantage of Pascal is the clarity of the code as opposed to some other languages such as FORTRAN or C. Pascal is not as efficient as these other languages, but in a prototype system like this clarity is a higher priority.

2. Pascal's Disadvantages are Commonly Known

Pascal's disadvantages for this particular implementation are no different than any other; however there are two that deserve special mention. Pascal input-output facilities afe very awkward to use. Any type of translating program, whether it be an interpreter or compiler, must scan an input program, either from a file or terminal, before execution. Pascal provides only for input to be read in one character at a time. This technique is obviously very inefficient, especially since it is well known that a good method of improving program efficiency is by reducing the number of I/O calls required. At a minimum, a better language would allow at least an identifier at a time to be read, while the ideal language would allow a large amount of data to be read into a buffer, which could then be scanned and used as needed, only more efficiently because it is in main memory. If the interpreter were reading from a disk, a logical amount of data to be read at one time would be an entire track.

C. THE CELL AND THE REPRESENTATION OF ATOMS AND LISTS

1. Atoms

In ELC, as in LISP, there are only two elements, atoms and lists. Atoms are non-divisible entities such as integers, real numbers, characters, and identifiers. For a complete breakdown of atoms and the rest of the language

II. INTERPRETER OPERATIONS

A. ASSUMPTIONS

It is assumed that the reader has a working knowledge of recursion and recursive languages, in particular Pascal. If not refer tc Cooper [Ref. 9] fcr information. The reader must also have knowledge of the Extended Lambda Calculus as presented by MacLennan [Ref. 6]. Complete descriptions of these areas are beyond the scope of this report.

The interpreter is a prototype system, so priorities were given to successful operation and to clarity of code rather than to efficiency. Efficiency was not completely forgotten and suggestions on future improvements in this area are given in Chapter 4, Conclusions.

B. PASCAL IS THE IMPLEMENTATION LANGUAGE

Pascal was chosen as the implementation language for the interpreter for two reasons. First, Pascal is the high level language taught to Computer Science students at the Naval Postgraduate School. Implementation of the interpreter in Pascal will thus facilitate its future use by students without the necessity of learning a new language. Second, using Pascal demonstrates that an interpreter of this type can be written in almost any programming language providing that it has recursion. Pascal, however, is not the ideal language for this type of project.

1. Pascal Does Have Some Advantages

The principal advantage of Pascal is the ability to dynamically allocate memory, which takes the burden of managing an array or heap space away from the programmer.

infinite loops. For examples of such programs see MacLennan [Ref. 6].

Is the halting problem a reason to disregard the value of functional progamming? If it is, then all other programming languages should be discarded. It is not unusual for programmers, using conventional languages, to occasionally write programs that go into infinite loops.

Finally, another reason why functional programming languages are not currently popular is that they do not work very efficiently on conventional architectures. As stated previously, most current architectures are von Neumann in nature, meaning they are sequential. The result is that the inherent parallelism of the functional languages cannot be exploited. There is work being done to design new architectures, some specifically for functional languages. One of the most promising is the reduction architecture proposed by Mago, which is described in [Ref. 8].

```
+:<1,2>
FP

sum[1 2]
KRC

<call <var sum> <con 1> <con 2>>
ELC
```

Figure 1.2 Sum in Three Functional Languages.

are the other implementations more readable because of their conciseness? The point is that readability means different things to different people. It also depends to a certain degree on training. A programmer well versed in FP will undoubtedly feel comfortable with the FP version and might find the ELC notation too verbose and wasteful. On the other hand, a person not familiar with FP or KRC may be able to tell more about what the function is supposed to do by reading the ELC version.

The conclusion is that the readability issue is not a good reason to abandon functional programming. Of course there is a certain amount of learning time required, as with any language, and it may even be more severe in this case due to the mathematical nature of these languages. However, if the benefits of exploiting parallelism and decreasing software maintenance costs can be achieved, they will far outweigh the disadvantage of a longer learning period.

Another problem area that has kept the popularity of functional languages to a minimum is the halting problem as described by MacLennan [**Ref. 6**]. As stated, since functional programs are constructed from expressions, evaluation order does not matter. This is true, however, only for problems that halt. It is possible to write some functional programs in such an order that will cause them to go into

```
Def! = eq0 --> 1: x dot (.id,! dot sub1.)
```

Note: dot = composition

FP

```
fact:
   proc(n) recursive returns(fixed);
      dcl (n, v) fixed;
      if n < 2
         then v = 1;
         else v = n * fact(n - 1);
      return(v);
   end fact;
```

PL/I

Figure 1.1 Factorial Program in FP and PL/I.

is easier to prove functional programs correct, software maintenance costs could improve dramatically. Early's project also demonstrated this fact in that the assembler written in C took sixty hours to complete compared to twenty for the FP version. One of the main reasons for this fact was that debugging time for the FP version was negligble. This was attributed to the fact that FP programs do what you expect of them since they are so easily proven correct. Functional languages are not without their critics and problems, however, a fact which merits discussion.

There are those that will argue that functional languages should not be used because they are not readable. This varies somewhat depending on the functional language being studied. In Figure 1.2 there are three examples of functions to take the sum of two numbers. They are written in Backus' FP, Turner's KRC, and ELC.

What does "readable" really mean? Is the ELC function more readable because it is obvious that a function is being called (because of the explicit use of the word "call"), or

```
function eval (e, a: list): list;
   var T,C,e1p: list;
       e1:alfa;
   begin
   if atomp(e) then eval := e
   else begin
      e1p := first(e);
      e1 := e1p@.aval;
      if e1 = 'list' then eval := evlis(rest(e), a)
      else if e1 = 'rinset' then eval := e
      else if e1 = 'con' then eval := first( rest(e))
      else if e1 = 'var' then eval := assoc( a,
         first( rest(e)))
      else if e1 = 'letrec' then eval :=
         letrec(first( rest(e)),
         first(rest(rest(e))),
         first(rest(rest(rest(e)))), a)
      else if e1 = 'lambda' then begin
         new(C, alf);
         cellcount(1, 'eval');
         with C@ do begin
            tag := alf;
            aval := 'closure';
            end;
         eval := cons( cons(C,e), cons(a,nil) );
         end {if e1 = 'lambda'}
      else if e1 ='if' then eval := evcon( rest(e), a)
      else if e1 = 'call' then
         eval := apply(eval(first(rest(e)),a),
           evlis(rest(rest(e)),a))
      else if e1 = 'apply' then
         eval:=apply(eval(first(rest(e)),a),
           eval(first(rest(rest(e))),a))
      else if e1 = 'let' then begin
         {First, evaluate actual parameters and then
          form the environment of evaluation for the
          let statement}
         T:= pairlis(evlis(first( first( rest(e))), a),
           evlis( first(rest( first( rest(e)))), a), a);
         eval :=
           eval(first(rest(rest (first(rest(e))))), T);
         end
      else errormsg('eval');
      end
   end {function eval};

                '@' Indicates pointer
```

Figure 2.6 Function Eval.

<list a b c>

the rest of the list or <a b c> is returned.

- con 'Con' tells the interpreter that the remainder of the list is a constant, so it is returned as such. Examples are:

```
< <first <prim first>> <rest <prim rest>>. . . >
```

Figure 2.7 Primitives Association List.

```
<con 1> = 1
<con <1 2 3>> = <1 2 3>
```

- var The keyword var, tells the interpreter to search the current environment of execution for the value of a certain bound variable. For example, if <var x> was sent to eval and the association looked like

  ```
  <<x 5> <y 'Navy'> <z 1400>. . .>
  ```

  eval would return the value of 5 for x. The search of the association list is performed by the assoc function of the interpreter. Refer to the source code for the interpreter found in Appendix C for a more detailed discussion of the assoc function.

```
<lambda <x> <call <var sum> <var x> <var x>>>
```

Figure 2.8 Lambda Expression.

- lambda Lambda expressions are ELC's analog to the procedure of conventional programming languages. These expressions are templates for solving certain problems using variables that must be bound to actual values before evaluation can take place. This template is commonly known as an abstraction. The example given in

2.8 is a lambda expression that can take any x, where x is an integer or real number, and add it to itself. It is currently not executable because no actual value for x is present. Since evaluation cannot be completed until later, the interpreter prepares the lambda expression for future execution by forming a closure. This is accomplished by using the primitive cons to add the keyword 'closure' to the front of the lambda expression and then using cons once again to add this to the front of the current environment, which has been placed in a list by itself. All that is left is to bind x with a value and add that to the current environment for execution to take place. This is accomplished by the apply function, which is triggered by the keyword 'call'.

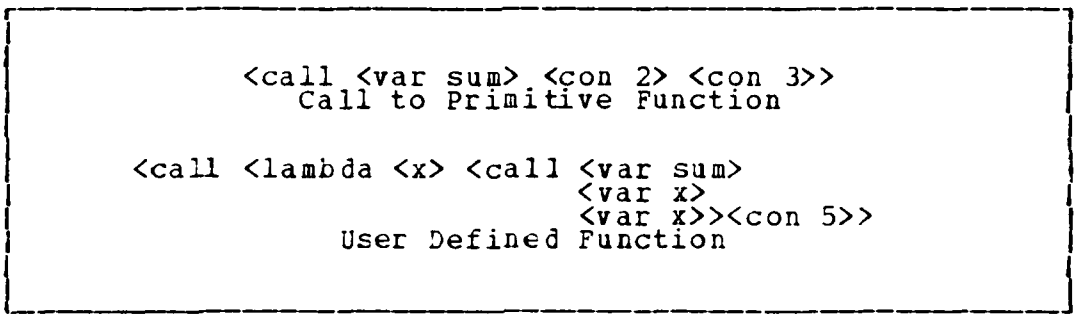


Figure 2.9 Using Call to Invoke Functions.

- call The keyword 'call' evokes the interpreter function apply to evaluate ELC function calls. The two simple examples given in Figure 2.9 are of a direct call to a primitive function and a call of the lambda expression discussed in the last section with the actual value of 5. The execution of each is traced below. Refer to Figure 2.6 to follow the trace.

<call <var sum> <con 2> <con 3>>

- 'call' is recognized by eval
- <var sum> is sent to eval with the current environment
- 'var' is recognized by eval
- 'sum' is looked up in the current environment by function assoc and <prim sum> is returned.
- The rest of the rest of the expression, which is <<con 2> <con 3>> is sent to function evlis which in turn sends each of the elements of the list to function eval with the current environment.Evlis returns a list of the results. In this case <2 3> is sent to function apply as the actual parameter.
- Function apply takes a function and applies it to a certain number of arguments. It acts somewhat like eval in that it strips off the first element of the list sent to it to determine how to precede. Since the first element is 'prim' the interpreter knows that the following element is the name of a primitive function. The result is that the function name, 'sum', and the arguments are sent to another function applyprim for final evaluation.
- sum, <2 3> are sent to applyprim.
- 2, 3 are sent to function sum.
- A pointer to the answer is sent back eventually reaching the initial call of function eval, the answer is printed, and evaluation stops.

- Note: The recursive nature of the interpreter is now clear, even for such a simple program. This point affects efficiency and should be considered as an area for future improvement.

<call <lambda <x> <call <var sum>
<var x>
<var x>>> <con 5>>

- 'call' recognized

- <lambda <x> <call <var sum> <var x> <var x>>> sent to eval with current environment.

- 'lambda' recognized and a closure is formed as follows <<closure lambda <x> <call <var sum> <var x> <var x>>> a > where a is the current environment.

- <con 5> is sent tc eval; 5 is returned.

- The closure and 5 are sent to function apply.

- In function apply 'closure' is recognized, so the body of the function call <call <var sum> <var x> <var x>> is sent to function eval. But an environment must be created before evaluation can be completed.

- x, 5, and the current environment is sent to function pairlis where the new environment is created by forming an attribute value pair of x and 5, <x 5>, and adding this to the current environment.

- Evaluation of the function now proceeds as the first example, except when <var x> is sent to eval the new environment is searched finding the value 5.

The consistency between how primitive and user defined functions are evaluated is now clear. This regularity aids the programmer because only one convention must be remembered to invcke all functions.

```
<if <<call <var equal> <con 0> <con 3>>
    <con 'true'>
    <ccn 'false'>>>
```

Figure 2.10 EIC Conditional.

• if The keyword 'if' signals that the remainder of the list is a conditional statement, so the rest of the list is sent to function evcon, which first determines the value of the first sublist, which must in turn be a call to one of the Boolean functions. In Figure 2.10,

<call <var equal> <con 0> <con 3>>

is the condition. Function evcon sends the conditional to function eval with the current environment of evaluation. If the condition is not a Boolean function call an error will cccur. If the condition evaluates to true, the result of evcon is the evaluation of the next sublist by eval. If the condition is false, the result is the evaluation cf the last sublist. In the example, since 0 and 3 are not equal the conditon is false so the last sublist is sent to eval, resulting in the constant 'false' being returned.

• Letrec The keyword 'letrec' is a signal to create a special environment fcr the evaluation of a recursive function. See Figure 2.11. There are four elements that must be considered:

- function name In this case 'append'

- lambda expression The abstraction
- body of the letrec Call to the function itself or another letrec expression.
- current environment

As with the normal function call a closure is formed and this is added to the front of the current environment. The difference is that the environment part of the closure points back to the point where the closure was inserted in the current environment. This is done so the function can be recursively called if need be or other var parameters can be locked up in the environment. See Figure 2.11 to see how the environment for the append function is constructed. As stated, letrec statements can be nested by including them as the body of another letrec, thus allowing the programmer to call any of the recursive functions above it in the body of the last recursive function. These functions can only look back and not forward. For examples see Appendix B, Sample Programs.

- Let The 'let' statement is simply a sugared version of the lambda statement and is included for clarity. Instead of writing

```
<call <lambda<x> <call <var sum>
                       <var x>
                       <var x>>> <con 5>>
```

the let statement allows you to write the expression in Figure 2.12. In general, <let <<x...> <y...> <B>>> means let x equal y in expression B. Any number of arguments can be included in the lists beginning with x and y. This type expression is particularly valuable if you want to assign a user defined function (lambda expression) a name which can then be called at any time. Consider the doubling function in Figure 2.12. The doubling function could have been

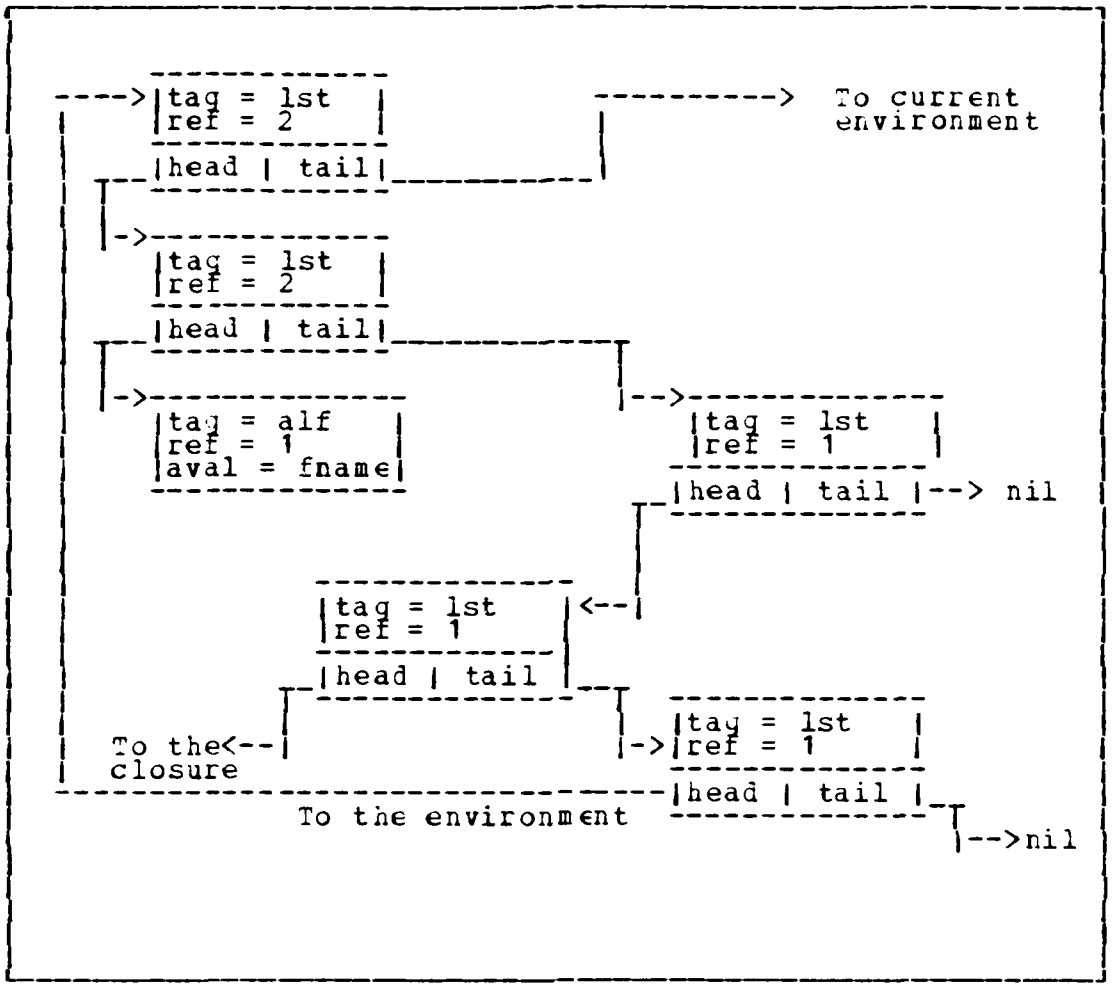


Figure 2.11 Letrec Environment.

accomplished using a single lambda expression as in Figure 2.8, but the let statement makes the function call expression

<call <var double> <con 2>>

clearer. Once again, expressions can be nested by inserting another 'let' statement for the B expression or even a 'letrec' statement. Examples are given in Appendix C, Sample Programs.

```
<let <<double> <lambda <x>
                       <call <var sum>
                             <var x>
                             <var x>>>
                       <call <var double>
                             <con2>>>>
```

Figure 2.12 Doubling Function.

• apply The keyword 'apply' triggers much the same action as 'call', except the arguments to the function are placed in a separate list as in

<apply <var sum> <list <con 2> <con 3>>>

The reason this feature is included is that some of the useful ELC programs require that arguments be reversed before functions are applied to them. This can only be accomplished if they are placed in a list so a recursive function call can reverse the elements. There is no primitive function included to handle this situation.

2. Printing Results

After function eval has completed evaluation, the result is in a tree form exactly like that described for the program itself. A pointer to the top of this tree is passed to procedure printval, which simply walks the tree and prints the information found in the leaves. This is done by checking the tags of the cells. If a cell's tag is 'lst', there is no information in the cell, only head and tail pointers. Since it is a list a left bracket must be printed. At that point the left and right cells are sent to printval recursively until a cell other than a 'lst' cell is found. These cells are obviously leaves of the tree so the variant portion of the cell is printed. This continues

until a tail pointer of one of the 'lst' cells is nil. This signals the end of the list sc a right bracket is printed and evaluation is completed.

F. MEMORY MANAGEMENT

1. Overview

Throughout the execution of a program many of the cells that are created become useless because they can no longer be accessed. Good examples of this are any of the binary functions included in the interpreter as primitives. For example, consider the sum function, Figure 2.13. Notice that two pointers are delivered to the sum function which point to the cells that contain the numbers to be added. After these numbers ar added, the results are placed in another cell. The two cells that held the intermediate results are no longer needed and should be returned to a free list to be used again latter. Another example is the creation of new environments for lambda expressions before they are evaluated. After the evaluation of the lambda expression, the cells that made up the attribute value pair that was added to the current environment are no longer needed and should be returned.

Since this is a prototype system, reference counting was chosen as the memory management method because of its simplicity and ease of installation. The model followed is outlined by MacLennan in [Ref. 6]. Reference counts refer to the number of pointers that a particular cell has referencing it at any one time. This count is kept in an additional field in each cell. Refer to Figure 2.4 to see the reference counts for a simple list. Reference counts must be incremented if additional references to cells are made. Reference counts in cells must be decremented if references are destroyed. References can be destroyed by overwriting

```
function sum(x, y: list) : list;
   var R,I: list;
   begin
      if (x@.tag=int) and (y@.tag=int) then begin
         if empty then begin
            new(I, int);
            cellcount(1, 'sum');
            end
         else
            I := freecell;
         with I@ do begin
            ref := 0;
            tag := int;
            I@.ival := x@.ival + y@.ival;
            end;
         sum := I;
         end
      else if (x@.tag=rea) and (y@.tag=rea) then begin
         if empty then begin
            new(R, rea);
            cellcount(1, 'sum');
            end
         else
            R := freecell;
         with R@ do begin
            ref := 0;
            tag := rea;
            R@.rval := x@.rval + y@.rval;
            end;
         sum := R;
         end
      else
         errormsg('sum') {Type mismatch}
   end; {Function sum}

                '@' Indicates Pointer
```

Figure 2.13 Function Sum.

pointers with other pointers by using an assignment statement or if the cell containing the pointer itself becomes inaccessible. Whenever a cell's reference count becomes zero it can be returned to the system because it is no longer accessible by the program.

The interpreter manages reference counts through the use of four procedures:

- ptrassn Overwrites pointers
- decr Decrements cell reference counts

- return Returns cells to the free list
- freecell Retrieves cells from the freelist

Figure 2.14 shows how function ptrassn works.

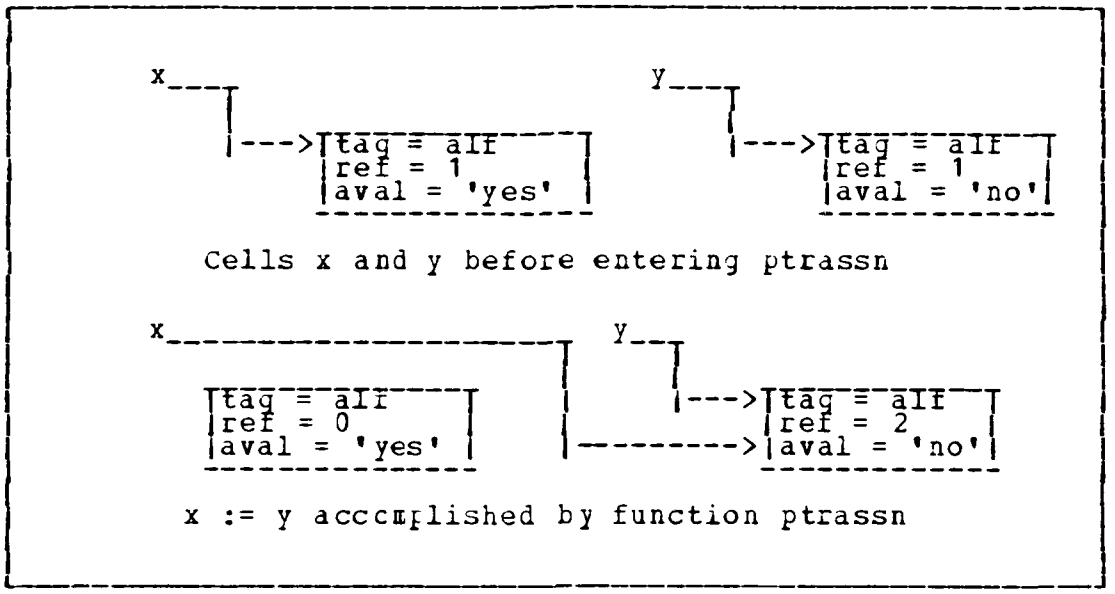


Figure 2.14 Function Ptrassn.

First, the reference count of the cell that x points to is decremented. Next the reference count of y is incremented. Finally x is assigned the value of y. It is important that the assignment statement be done last so the reference count of x can be decremented. If not done x would no longer point to the correct cell and the reference count of y would actually be decremented.

Procedure decr is used to decrement the reference counts of all cells. If a reference count of a cell goes to zero, decr is recursively called over the entire list until all cells with references of zero are found and returned to a freelist maintained by procedure return.

Procedure return links all the free cells together by first making them all 'lst' cells and linking them through the tail fields with the tail field of the last cell in the list being set to nil.

Freecell is a procedure that is used to recover cells from the freelist instead of creating newcells by using the Pascal new facility. Each location in the interpreter that needs to create new cells first checks to see if the freelist is empty. If it is not, a cell is taken from the freelist instead of creating a new one. Actually a freelist is not necessary. Cells could be returned to the system using Pascal's dispose feature. Since this is a prototype system, the freelist is maintained to make it easier to maintain statistics on the number of cells being returned.

2. Conventions

a. Cell Creation

The reference counts of cells are set to zero when they are created. This must be done so that when a program is read into the interpreter reference counts in all cells are set to one. To understand this, study the cons function which is used to build up the program list when it is initially read. Cons uses the ptrassn procedure to set the head and tail of the connecting cells. If a cell is created during readin and its reference count is set to one, that reference count will go to two when that cell is sent to cons. The result is a reference count that is greater than it should be. If, however, the cell is created with a reference count of zero, it will be set to one when it is sent to cons, which in turn sets the head and tail of the connecting cell with the ptrassn procedure. The only special case that must be recognized is the cell at the very

top of the program tree which must be physically set to one after readin since it is never sent to the ptrassn procedure.

b. Local Declarations

If locally declared pointer variables are used to overwrite other pointers their reference counts must be decremented before the procedure they are declared in is completed. This is done because locally declared variables are only visible within the procedures they are declared in and then destroyed. If the reference counts they genereated are not decremented, excess reference counts to some cells are the result.

c. Passed Parameters

The reference counts of cells referenced by pointer variables passed to procedures or functions by value must be incremented upon entering the procedure and decremented when leaving the procedure. It is easy to see how cells can be recovered in this manner. If the reference count of a cell is zero when it enters a procedure it will be incremented to one during the execution of the procedure and then decremented to zero and reclaimed when the procedure is finished.

Function Reverse

Purpose

Takes any list as an argument, reverses the elements of the list and places them in another list.

Practical Application

Reverse is used primarily as a sub-function for larger programs. It is the nature of recursion that many times result lists are constructed in reverse order. The reverse function is then needed to regain the proper order.

Discussion

There are two versions of reverse included, reverse and revaux. Reverse makes use of the primitive 'conr' to build the result list where revaux utilizes a null list, (<>), to build the result list using a series of calls to 'cons'. It is interesting to study the differences in efficiency between the two functions. Reverse is faster and uses less memory. The reason is because the primitive conr was included in the interpreter, which shortens the number of steps required. Whether time and memory savings justify including another primitive in the interpreter depend on how often it is used. The use of the reverse function is minimal and would not justify including a primitive only for its use.

Source Code

```
<letrec reverse
   <lambda <L>
      <if <<call <var null> <var L>>
          <con <>>
          <call <var conr>
                <call <var reverse>
                      <call <var rest> <var L>>>
                <call <var sub> <var L> <con 1>>>>>>>
```

702	83	readval
704	108	nonblank
714	25	readlist
770	1	readint
827	199	digit
835	257	letter
843	57	readident
867	83	readval
879	11	evcon
880	157	evlis
881	52	apply
882	1	letrec
884	190	eval
947	1	letrec
1012	11	evcon
1041	157	evlis
1093	41	applyprim
1198	52	apply
1239	28	dcprim
1261	1	readfname

```
                    Statistics
   System time was        366 milliseconds
   User time was         6600 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                   84   |
|cons                    402   |
|readiden                 57   |
|readint                   1   |
|letrec                    6   |
|null                     11   |
--------------------------------
 Total cells            561

Profile for append function
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984  test11.p
Profiled Thu Dec 13 09:05 1984

       Line       Count

          1           1           func
         33          21                printval
         36         505                cellcount
         63        1477                nullp
         84        3123                first
         94        1692                rest
        122         402                cons
        258          10                sub
        539         190                atomp
        547        1477                nullp
        564          11                null
        599          94                assoc
        643          33                pairlis
        663          21                printval
```

```
<call <var append>
      <list <con g><con h> <con j>>
      <list <con q> <con r><con s>>>>!
```

Results of Append Function (Compiled Interpreter)

Enter Expression

<a, b, c, d, e, f, g, h, i, j, j, l, m, n, o, p, q, r, s, t>

Evaluation Completed

****

Statistics

System time was 183 milliseconds

User time was 616 milliseconds

Module	Cells created
dcprim	84
cons	402
readiden	57
readint	1
letrec	6
null	11
Total cells	561

Results of Append (EIC Interpreter interpreted by Berkeley Pascal)

Enter Expression

<a, b, c, d, e, f, g, h, i, j, j, l, m, n, o, p, q, r, s, t>

Evaluation Completed

****

APPENDIX B
SAMPLE PROGRAMS

General

The statistics in this appendix are referred to as compiled versus interpreted. This means compiled and interpreted versions of the ELC interpreter. Also all statistics refer to programs run on an interpreter without a memory manager. Run times are much slower when the memory management system is used.

Function Append

Purpose

The append function concatenates lists. This is different than the primitive cons which makes its first argument the first element of another list.

Practical Application

Append could be useful if the argument lists were large databases that had to be combined. This is common practice in database work where many small databases are combined to form a whole.

Source Code

```
<letrec append
   <lambda <L M>
      <if <<call <var null> <var L>>
            <var M>
            <call <var cons>
                  <call <var sub><var L><con 1>>
                  <call <var append>
                        <call <var rest><var L>>
                        <var M>> >>>>
```

```
<conditional>::='<'if<boolean exp.>
                    <list>|<lookup var>|<const exp.>|
                    <prim applic.>|<boolean exp.>
                    <list>|<lookup var>|<const exp.>|
                    <prim applic.>|<boolean exp.>

<boolean exp.> ::= '<call <var' <boolean prim> '>>'

<boolean prim> ::= atom|null|equal|memb|GT|LT|GE|LE

<prim call> ::= '<var' <primname> '>'

<primname>::=first|rest|cons|atcm|null|sum|subt|prod|divi|sub|
            equal|len|memb|repr|GT|LT|LE|GE

<prim applic.>::='<call'<prim call><list>ee1|
                 <lookup var>ee1|<const exp.>ee1 '>'

<list> ::= '< list'  <letter>ee1 | <letter>ee1 <list>ee1|
                     <number>ee1    <number>ee1

                         <list>ee1 <letter> ee1
                                   <number> ee1 '>'

<lookup var> ::= '< var' <letter>|<identifier> '>'

<const exp.> ::= '< con' <number>ee1 | <list> | <letter> '>'

<actuals> ::= <list> | <lookup var> | <const exp.> |

                <lookup var>      <lookup var>
                <list>       ee1  <list>       ee1
                <const. exp.>     <const. exp.>

<letter> ::= <a..z|A..Z>

<number> ::= <digit>ee1|<digit>ee1 '.' <digit>ee1

<digit> ::= <0..9>

<atom> :: letter|number|identifier
```

APPENDIX A
ELC GRAMMAR

Note: ' ' denotes literal copy.
ee denotes superscript. (ee1 means one or more>

<ELC program> ::= atom|list|<recursive exp.>|<abstraction>|
<application>|<let> '!'

<recursive exp.> ::= '< letrec' <rec identifier>
<lambda exp.> <body> '>'

<body> ::= '<call ' <rec identifier> <actuals>|
<recursive exp.>|<let> '>'

<rec identifier> ::= <identifier>

<identifier> ::= <letter> ee10|<<letter><number|letter>>ee10
Note: Ten or less letters. Corresponds to the Berkeley Pascal built in string, packed array 1..10 of char.

<abstraction> ::= '< lambda' <bound variables>
<abstraction body> '>'

<let> ::= '<let<'<bound variables> '<'
<abstraction>|<application>|
<primitive application >|
<conditional>|<recursive exp.> '>'

<application> ::= '<' <lambda exp.> <actuals>ee1 '>'

<bound variables> ::= <letter> ee1| <identifier> ee1

<abstraction body>::='<'<conditional>|<prim call>|
<abstraction>'>'

data must be inserted in the program itself. This could be done easily by modifying the interpreter to recognize key words that trigger a read operation.

Finally, the interpreter should be written in a more portable version of Pascal. Berkeley Pascal has several non standard features such as the alfa type that make its code machine dependent.

D. LESSONS LEARNED

Writing programs in ELC becomes easier with experience. This was primarily because detailed programs are built by combining several smaller programs. For example, the program that generates the trig table is made of six functions, each a program in its own right. Once the single function programs are tested, they can be easily and reliably used to build other programs.

ELC programs also force the user to think about problems as a whole when programming. For example, when writing the append function one asks the question, "How would I physically solve this problem?". The answer is by taking one element at a time from one list and adding that element to the second list until the first list is empty. That explanation is exactly how to solve the problem recursively and the ELC program reflects that. If a conventional language was used to solve the problem, however, the programmer would have to be concerned with many low level constructs such as assignment statements and counter variables. After some experience, dealing with problems at a higher level became very comfortable, particularly because many of the problems encountered were solved using the same technique.

B. STRUCTURING PROGRAMS

Programs are contained in one list so they can be written in one line, but this does not always present a clear view of what the program does. A natural method of structuring ELC programs evolved through experience. The method is to stack the arguments of functions under their calls. For example, consider this call to the primitive function cons.

```
<call <var cons>
      <list a b>
      <list c d e>>
```

This convention becomes very useful in large programs when many functions must be nested. The conditional can be structured as

```
<if <<call <boolean exp.>
     <True consequent>
     <False consequent>>>
```

Once again the arguments are stacked for clarity.

C. FUTURE IMPROVEMENTS

There are several improvements that can immediately be accomplished for the interpreter.

The code could be made more English like. This could be done by writing a front end to translate a higher level code into the ELC code used in this report or by completely changing the ELC grammar. While making more readable programs this feature would decrease efficiency.

In the opposite direction the code could be made more mathematical in nature, similar to the notation used by Backus in FP. The tradeoff in that case would be efficiency versus readability.

A feature should also be added to allow data for the ELC programs to be read from the terminal or a file. Currently

these values and subscript them based on the number of scoping lines crossed in gettin from a use of the variable to its definition. A detailed explanation of this method is given by MacLennan [Ref. 6]. The beauty of this method is that it eliminates the overhead of managing the pointers of the association list and the need to recursively search it, both very expensive operations in terms of efficiency.

Comparisons are also made in Appendix B between two recursive ELC programs and their Pascal counterparts. The programs calculate n factorial and generate the first n elements of the Fibonacci sequence. The Pascal programs run faster, which is not suprising because there is one less layer of software invloved in their execution. The time differences are less than a second, however, and with minimal improvements to the interpreter can be improved.

Finally, a more efficient memory managing system should be implemented. Programs executed with the memory manager are very slow as can be seen by comparing the run times of the programs listed on the last page of Appendix B with their execution times without the memory manager. A mark and sweep system would be more efficient because the execution of a program would not be impeded unless all the allocated memory was used. On the other hand, the reference counting system invokes memory management procedures and functions throughout program execution. Since most programs will not use all allocated memory they would run at near normal speed (normal speed being the time to execute a program without the memory manager). The tradeoff is that the interpreter will have to allocate its own heap space and manage it.

IV. CONCLUSIONS

A. EFFICIENCY

Appendix B contains statistical data for several ELC programs. Times for program execution are given for both interpreted and compiled versions of the interpreter. It is not suprising that the compiled version always ran much faster and is recommended for use. The interpreted version of the ELC interpreter was used throughout development, however, because it took half the time that compiling required.

Profiles for all sample programs are also included in Appendix C. These profiles reveal hints on how the interpreter could be more efficient. The data shows that the interpreter spends most of its time in the primitive functions, such as null, first, sum, etc.. Efficiency could be improved by writing these functions in a lower level language, such as assembly language, and then linking these modules in at run time. This would not be difficult because these functions are very short and they are all constructed in the same manner, e.g., all the Boolean functions are the same except for the condition being checked. When the lower level code is completed for one of the modules it could be used as a template for the others.

Efficiency can also be improved by replacing the association list mechanism for looking up the value of variables. Since the pairlis function always adds new attribute value pairs to the front of the current environment before evaluation, it is clear that searching an association list is not always necessary because we know the value is at the front of the list. A better method is to use an array to hold

C. ERROR MESSAGES

The best way to become familiar with the interpreter's error messages is to study the error handling procedure of the interpreter itself, Appendix C. The procedure is set up like a table displaying all the error messages and they can be easily traced back to their sources. The interpreter is designed to halt exection immediately upon detection of an error.

At this point programs can be typed directly from the terminal and executed. To stop execution follow the last program with a '!'. A statistical summary is given showing the number of cells created, number of cells returned to the system, and time of execution before termination. Evaluation is successfully completed with the message

Evaluation Completed.

If a long program is to be executed, it is recommended to place it in a file so editing can be accomplished, if necessary. The command to interpret a program from a file is

obj

The interpreter then responds with a prompt to ask for the name of the file where the program exists.

File for ELC Program:

The filename can be up to eighty characters in length.

B. EXECUTION TRACE

After the method of loading the program is determined, the user is questioned if a trace is desired. A trace prints out pertinent intermediate results as the program is executed to help in debugging. Two examples of items printed out are: each expression sent to function eval and results of looking up a var parameter in an association list. Not all expressions can be printed out because some structures are recursive and an attempt to print them out results in an infinite loop. To avoid infinite loops additional questions are asked about the user's desire to print out certain structures when there is a possibility that they could be recursive. Invoking the trace facility obviously slows execution a great deal but can be quite helpful in debugging a program.

III. HUMAN INTERFACE WITH THE INTERPRETER

A. LOADING A PROGRAM

The interpreter is activated by first compiling or interpreting it using the facilities of Berkeley Pascal under Unix 4.2 BSD as shown in the following example:

Interpreted Code
'pi' <filename of the interpreter>'.p'

Compiled Code
'pc' <filename of the interpreter>'.p'

If interpreted, an executable file named 'obj' is created; if compiled, an executable file, 'a.out' is created. The complied version runs much faster as seen by the time of execution statistics located in Appendix B, Sample Programs. It is recommended that the names of these files be changed to something more intuitive, such as 'ELC' or 'Interp', etc..

The interpreter can be run in interactive mode or a program can be executed from ancther file. Interactive mode should only be used for short programs or if the interpreter is being used as a calculator to perform basic mathematical computations. The big drawback to interactive use is that no editing can be done on programs that are longer than one line when typing at the terminal. If interactive mode is desired, the following command should be issued. Interpreted code is assumed in all examples.

obj i

The 'i' toggle tells the interpreter that interactive mode is desired. A logon message with date and time appears next, followed by the prompt:

Enter Expression

```
<call <var reverse> <list a b c d e f g h i j>>>>!

Results of reversing a ten element list (Compiled version)
Enter Expression
<j, i, h, g, f, e, d, c, b, a>
Evaluation Completed

**************************************************************
                        Statistics
System time was          100 milliseconds
User time was            516 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                  84    |
|cons                   323    |
|readiden                42    |
|readint                  1    |
|letrec                   6    |
|null                    11    |
|conr                     9    |
--------------------------------
 Total cells           476

Reversing a ten element list(interpreted)
Enter Expression
<j, i, h, g, f, e, d, c, b, a>
Evaluation Completed

**************************************************************
                        Statistics
System time was          316 milliseconds
User time was           5716 milliseconds
```

Module	Cells created
dcprim	84
cons	323
readiden	42
readint	1
letrec	6
null	11
conr	9
Total cells	476

Profile of reverse function
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984 test11.p
Profiled Thu Dec 13 09:08 1984

Line	Count	
1	1	func
33	11	printval
36	420	cellcount
63	1349	nullp
84	2861	first
94	1545	rest
122	323	cons
137	10	conr
258	10	sub
539	169	atomp
547	1349	nullp
564	11	null
599	83	assoc
643	22	pairlis
663	11	printval
702	67	readval

704	91	nonblank
714	24	readlist
770	1	readint
827	167	digit
835	210	letter
843	42	readident
867	67	readval
879	11	evcon
880	135	evlis
881	52	apply
882	1	letrec
884	169	eval
947	1	letrec
1012	11	evcon
1041	135	evlis
1093	41	applyprim
1198	52	apply
1239	28	dcprim
1261	1	readfname

Function Revaux

Source Code

```
<letrec revaux
   <lambda <L M>
      <if <<call <var null> <var L>>
          <var M>
          <call <var revaux>
                <call <var rest> <var L>>
                <call <var cons>
                      <call <var sub> <var L> <con 1>>
                      <var M>
                >>>>>
 <call <var revaux>
       <list a b c d e f g h i j>
       <con <>> >>!
```

Results of Revaux (10 element list, compiled)

Enter Expression

<j, i, h, g, f, e, d, c, b, a>

Evaluation Completed

****

Statistics

System time was 166 milliseconds

User time was 566 milliseconds

Module	Cells created
dcprim	84
cons	383
readiden	47
readint	1
letrec	6

```
|null                    11     |
--------------------------------
  Total cells             532

Results of revaux (interpreted).
Enter Expression
<j, i, h, g, f, e, d, c, b, a>

Evaluation Completed

****************************************************************
                        Statistics
System time was          366 milliseconds
User time was           6333 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                  84     |
|cons                   383     |
|readiden                47     |
|readint                  1     |
|letrec                   6     |
|null                    11     |
--------------------------------
  Total cells             532


Profile of Revaux
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984  test11.p
Profiled Fri Dec 14 22:46 1984

       Line       Ccunt

          1           1           func
         33          11                printval
         36         476                cellcount
```

63	1466	nullp
84	3104	first
94	1672	rest
122	383	cons
258	10	sub
539	180	atomp
547	1466	nullp
564	11	null
599	94	assoc
643	33	pairlis
702	74	readval
704	100	nonblank
714	26	readlist
770	1	readint
827	178	digit
835	226	letter
843	47	readident
867	74	readval
879	11	evcon
880	146	evlis
881	52	apply
882	1	letrec
884	180	eval
947	1	letrec
1012	11	evcon
1041	146	evlis
1093	41	applyprim
1198	52	apply
1239	28	dcprim
1261	1	readfname

Map Functional

Purpose

Functionals are functions that return other functions as results. The map functional allows the user to take any unary function and apply it to the elements of a list, returning a list of the results. In this example, the sine function is mapped across a list of ten angles.

Practical Application

Map could be used extensively in business applications. An example would be an employee database where the same operations must be accomplished on many different records. If salaries were increased across the board, a version of map could be used to achieve this.

Source Code

```
<letrec map
   <lambda <f>
      <lambda <L>
         <if <<call <var null> <var L>>
                <con <>>
                <call <var cons>
                    <call <var f> <call <var first><var L>>>
                    <call <call <var map> <var f>>
                           <call <var rest> <var L>>>>>>>>
 <call <call <var map> <var sin>>
          <list <con 45> <con 60> <con 90>>> >!

 Results of map functional (map sine) Compiled
 Enter Expression
 <0.707107, 0.866025, 1.000000, 0.913545, 0.573576,
  0.342020, 0.422618, 0.275637, 0.939693, 0.999391>

 Evaluation Completed
```

****

```
                        Statistics
System time was          216 milliseconds
User time was            683 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                   84   |
|cons                    445   |
|readiden                 52   |
|readint                  10   |
|letrec                    6   |
|eval                     11   |
|null                     11   |
|sinp                     10   |
--------------------------------
 Total cells            629

Map sine (interpreted)
Enter Expression
<0.707107, 0.866025, 1.000000, 0.913545, 0.573576,
 0.342020, 0.422618, 0.275637, 0.939693, 0.999391>

Evaluation Completed

****************************************************************
                        Statistics
System time was          416 milliseconds
User time was           7666 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                   84   |
|cons                    445   |
|readiden                 52   |
|readint                  10   |
```

letrec	6
eval	11
null	11
sinp	10
Total cells	629

Profile for map functional
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984 test11.p
Profiled Thu Dec 13 10:15 1984

Line	Count	
1	1	func
33	11	printval
36	573	cellcount
63	1666	nullp
84	3579	first
94	1959	rest
122	445	cons
411	10	sinp
539	212	atomp
547	1666	nullp
564	11	null
599	104	assoc
643	44	pairlis
663	11	printval
702	103	readval
704	144	nonblank
714	41	readlist
770	10	readint
827	217	digit
835	279	letter
843	52	readident

867	103	readval
879	11	evcon
880	167	evlis
881	73	apply
882	1	letrec
884	212	eval
947	1	letrec
1012	11	evcon
1041	167	evlis
1093	51	applyprim
1198	73	apply
1239	28	dcprim
1261	1	readfname

Halving Function

Purpose

The halving function takes a list of numbers and returns a list of all the elements divided in half.There is really no practical application for this function but it demonstrates the use of the 'bu' functional which changes a binary operator to a unary operator. If you divide a list of integers by 2 it is more efficient to fix the second operand of the division instead of evaluating 2 as a constant each time the division takes place.

```
<letrec map
   <lambda <f>
      <lambda <L>
         <if <<call <var null> <var L>>
                <con <>>
                <call <var cons>
                    <call <var f> <call <var first><var L>>>
                    <call <call <var map> <var f>>
                           <call <var rest> <var L>>>>>>>>
<let <<bu> <<lambda <f k>
                <lambda <x>
                   <call <var f> <var k> <var x>>>> >
<let <<revf> <<lambda <f>
                <lambda <x y>
                   <call <var f> <var y> <var x>>>>>
<call
  <call <var map>
        <call <var bu>
              <call <var revf> <var divi>>
              <con 2>>>
<list <con 4> <con 6> <con 8> <con 18>>
                                        >>>>>>!
```

Results of halving function(compiled)

```
Enter Expression
<      2,      3,      4,      9,      10,      11,      12,
      13,     14,     15>

Evaluation Completed

*****************************************************************
                          Statistics
System time was          183 milliseconds
User time was            966 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                  84    |
|cons                   674    |
|readiden                87    |
|readint                 11    |
|letrec                   6    |
|eval                    15    |
|null                    11    |
|divi                    10    |
--------------------------------
  Total cells             898

Results of halving (interpreted)
Enter Expression
<      2,      3,      4,      9,      10,      11,      12,
      13,     14,     15>

Evaluation Completed

*****************************************************************
                          Statistics
System time was          483 milliseconds
```

User time was 10183 milliseconds

Module	Cells created
dcprim	84
cons	674
readiden	87
readint	11
letrec	6
eval	15
null	11
divi	10
Total cells	898

Profile for halving function (use of bu).
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984 test11.p
Profiled Thu Dec 13 10:03 1984

Line	Count	
1	1	func
33	11	printval
36	842	cellcount
63	2057	mulip
84	4420	first
94	2494	rest
122	674	cons
233	10	divi
539	305	atomp
547	2057	nulip
564	11	null
599	166	assoc

643	103	pairlis
663	11	printval
702	168	readval
704	238	nonblank
714	70	readlist
770	11	readint
827	350	digit
835	448	letter
843	87	readident
867	168	readval
879	11	evcon
880	240	evlis
881	95	apply
882	1	letrec
884	305	eval
947	1	letrec
1012	11	evcon
1041	240	evlis
1093	51	applyprim
1198	95	apply
1239	28	dcprim
1261	1	readfname

```
|equal                  1806      |
|sum                     255      |
|len                     411      |
--------------------------------

  Total cells          57682
```

```
Profile for frequency table function.
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984  test11.p
Profiled Thu Dec 13 10:20 1984
```

Line	Ccount	
1	1	func
33	19	printval
36	57626	cellcount
62	1806	equal
63	664983	nullp
84	1404922	first
94	743026	rest
122	49559	cons
158	255	sum
273	2267	equalp
291	1806	equal
539	70799	atomp
547	664983	nullp
564	4554	null
576	1233	lenp
587	411	len
599	36403	assoc
643	10367	pairlis
663	19	printval
702	530	readval
704	744	nonblank
714	214	readlist

```
|null                    4554      |
|memt                     666      |
|equal                   1806      |
|sum                      255      |
|len                      411      |
--------------------------------

  Total cells          57682
```

Results of frequency table generator(interpreted)

```
Enter Expression
<<text,      1>
, <of,      1>
, <block,       1>
, <the,      1>
, <is,      1>
, <This,       1>
>
```

Evaluation Completed

****

Statistics

System time was 23900 milliseconds
User time was 1990866 milliseconds

```
--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                  84     |
|cons                 49559     |
|readiden                313     |
|readint                   3     |
|letrec                   30     |
|eval                      1     |
|null                   4554     |
|memt                    666     |
```

```
                                        <var T>>
                                   <call <var freq>
                                        <call <var rest>
                                             <var T>>>>
                                  <con 1>>
                              <ccn <>>>>>>>>
<call <var freq>
     <call <var repr>
         <finset This is the block of text>>>>>>>>>>!

Results of frequency table generator (compiled)

Enter Expression
<<text,       1>
, <of,       1>
, <block,        1>
, <the,       1>
, <is,       1>
, <This,       1>
>

Evaluation Completed

*************************************************************
                         Statistics
System time was        4916 milliseconds
User time was        157966 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                   84    |
|cons                  49559    |
|readiden                313    |
|readint                   3    |
|letrec                   30    |
|eval                      1    |
```

```
                <if <<call <var equal>
                           <var k>
                           <call <var first>
                                 <call <var first> <var T>>>>
                      <call <var first>
                           <call <var rest>
                                 <call <var first> <var T>>>>
                      <call <var lookup>
                           <call <var rest> <var T>>
                           <var k>>
                                      >>>>>
<let <<occur> <<lambda <w F>
                       <if <<call <var equal>
                                  <call <var memb>
                                        <var w>
                                        <call <var dom>
                                              <var F>>>
                                  <con true>>
                             <call <var lookup>
                                   <var F>
                                   <var w>>
                             <con 0>>>>>
<letrec freq
  <lambda <T>
     <if <<call <var null> <var T>>
          <con <>>
          <call <var overlay>
                <call <var freq>
                      <call <var rest> <var T>>>
                <call <var cons>
                      <call <var first> <var T>>
                      <call <var cons>
                            <call <var sum>
                                 <call <var occur>
                                       <call <var first>
```

```
                                        <var T>>>
                                  <call <var rest>
                                        <call <var dom>
                                        <var T>>>>
                              <con true>>
                        <con false>
                        <call <var isfinfunc>
                              <call <var rest>
                                    <var T>>>>>
                  <con false>>>>>>
<letrec overlay
  <lambda <T pr>
     <if <<call <var equal>
                 <call <var isfinfunc> <var T>>
                 <con true>>
           <if <<call <var null> <var T>>
                <call <var cons> <var pr> <con <>>>
                <if <<call <var equal>
                           <call <var first> <var pr>>
                           <call <var first>
                                 <call<var first>
                                      <var T>>>>
                     <call <var overlay>
                           <call <var rest> <var T>>
                           <var pr>>
                     <call <var cons>
                           <call <var first> <var T>>
                           <call <var overlay>
                                 <call <var rest> <var T>>
                                 <var pr>>>>>>>>
           <con Tnotffunc>>>>
<letrec lookup
  <lambda <T k>
     <if <<call <var null> <var T>>
          <con notfound>
```

Frequency Table Generator

Purpose

This program takes a finite set of text and creates a frequency table of the words used and how many times they are used.

Practical Application

This program could be useful if extended to recognize patterns in large blocks of data. Also, in military intelligence work, it could be vaulable to see how many times a persons name appears in a newspaper to gain some insight into how important they might be.

```
<letrec dom
   <lambda <L>
      <if <<call <var null> <var L>>
          <con <>>
          <call <var cons>
                <call <var first>
                      <call <var first> <var L>>>
                <call <var dom>
                      <call <var rest>
                            <var L>>>>>>>
 <letrec isfinfunc
   <lambda <T>
      <if <<call <var null> <var T>>
          <con true>
          <if <<call <var equal>
                     <call <var len>
                           <call <var first> <var T>>>
                     <con 2>>
              <if <<call <var equal>
                         <call <var memb>
                               <call <var first>
                                     <call <var dom>
```

94	1148	rest
122	395	cons
137	3	conr
258	3	sub
411	3	sinp
539	123	atomp
547	988	nullp
564	8	null
599	61	assoc
643	23	pairlis
663	4	printval
702	154	readval
704	219	nonblank
714	65	readlist
770	4	readint
827	377	digit
835	466	letter
843	85	readident
867	154	readval
879	8	evcon
880	96	evlis
881	39	apply
882	2	letrec
884	123	eval
947	2	letrec
1012	8	evcon
1041	96	evlis
1093	29	applyprim
1198	39	apply
1239	28	dcprim
1261	1	readfname

```
Results cf compositon (interpreted)
Enter Expression
<1.000000, 0.866025, 0.707107>

Evaluation Completed

************************************************************
                        Statistics
System time was          583 milliseconds
User time was           5966 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                   84   |
|cons                    395   |
|readiden                 85   |
|readint                   4   |
|letrec                   12   |
|eval                      2   |
|null                      8   |
|conr                      2   |
|sinp                      3   |
--------------------------------
 Total cells             595

Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984  test11.p
Profiled Mon Dec 17 18:51 1984

       Line       Count

          1           1           func
         33           4                printval
         36         539                cellcount
         63         988                nullp
         84        2111                first
```

```
                                <call <var rest >
                                      <var L >>> >>>>
<let <<dot> <<lambda <f1 f2>
              <lambda <x>
                 <call <var f1><call<var f2><var x>>>>>>
<call
     <call <var dot> <var mapsin> <var reverse>>
<list <con 45> <con 60> <con 90>> >>>>>!


Results of compositcn (compiled)

Enter Expression
<1.000000, 0.866025, 0.707107>

Evaluation Completed

************************************************************
                          Statistics
System time was          250 milliseconds
User time was            616 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                   84   |
|cons                    395   |
|readilen                 85   |
|readint                   4   |
|letrec                   12   |
|eval                      2   |
|null                      8   |
|conr                      2   |
|sinp                      3   |
--------------------------------
 Total cells             595
```

Composition Functional

Purpose

Composition allows the output of one function to act as the input to another function.

Practical Application

Composition could be used in business applications as a way of querying a database with multiple conditions. For example, utilizing the filter function, a user could ask for records of employees that satisfy a certain condition and then apply another call to filter with a further refined condition such as all employees in department 5 that make more than two thousand dollars a week. In this example mapsin and reverse are composed. The composition function is named dot to correspond to Backus's FP language which actually includes this as an operator in the language.

Source Code

```
<letrec reverse
   <lambda <L>
      <if <<call <var null> <var L>>
           <con <>>
           <call <var conr>
                 <call <var reverse>
                       <call <var rest> <var L>>>
                 <call <var sub> <var L> <con 1>> >>>>

 <letrec mapsin
   <lambda <L >
      <if <<call <var null > <var L >>
            <con <>>
            <call <var cons >
                  <call <var sin >
                        <call <var first ><var L >>>
                   <call <var mapsin >
```

663	12	printval
702	150	readval
704	205	nonblank
714	55	readlist
770	15	readint
827	332	digit
835	427	letter
843	80	readident
867	150	readval
879	32	evcon
880	260	evlis
881	93	apply
882	1	letrec
884	335	eval
947	1	letrec
1012	32	evcon
1041	260	evlis
1093	82	applyprim
1198	93	apply
1239	28	dcprim
1261	1	readfname

Module	Cells created
dcprim	84
cons	531
readiden	80
readint	15
letrec	6
null	22
LE	10

Total cells 748

Profile for collate function
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984 test11.p
Profiled Thu Dec 13 10:07 1984

Line	Count	
1	1	func
33	12	printval
36	692	cellcount
63	2566	nullp
84	5526	first
94	2921	rest
122	531	cons
258	30	sub
384	10	LEp
394	10	LE
539	335	atomp
547	2566	nullp
564	22	null
599	166	assoc
643	33	pairlis

```
<      2,      3,      3,      5,      6,      6,      7,
       8,      9,     10,     12>

Evaluation Completed

**************************************************************
                        Statistics
System time was          166 milliseconds
User time was           1066 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                  84     |
|cons                   531     |
|readiden                80     |
|readint                 15     |
|letrec                   6     |
|null                    22     |
|LE                      10     |
--------------------------------
 Total cells            748


Collate function (interpreted)
Enter Expression
<      2,      3,      3,      5,      6,      6,      7,
       8,      9,     10,     12>

Evaluation Completed

**************************************************************
                        Statistics
System time was          633 milliseconds
User time was          11316 milliseconds
```

Collating function

Purpose

Collate takes two sorted lists and merges them into one sorted list.

Practical Application

Sorting and collating are standard office functions that benefit from automation. A sorting function needs to be combined with collate to initially sort the sublists.

Source Code

```
<letrec collate
   <lambda <L M>
      <if <<call <var null> <var L>>
          <var M>
          <if <<call <var null> <var M>>
              <var L>
              <if <<call <var LE>
                         <call <var sub> <var L> <con 1>>
                         <call <var sub> <var M> <con 1>>>
                  <call <var cons>
                         <call <var sub> <var L> <con 1>>
                         <call <var collate>
                               <call <var rest> <var L>>
                               <var M>>>
                  <call <var cons>
                         <call <var sub> <var M> <con 1>>
                         <call <var collate>
                               <var L>
                               <call<var rest>
                                    <var M>>>>>>>>>>>>
<call<var collate><list 2 5 6 8 12><list 3 3 6 7 9 10>>>!
```

Result of Collate Function, Compiled

Enter Expression

770	3	readint
827	1393	digit
835	1709	letter
843	313	readident
867	530	readval
879	6360	evcon
880	53265	evlis
881	23613	apply
882	5	letrec
884	70799	eval
947	5	letrec
1012	6360	evcon
1041	53265	evlis
1061	1115	membp
1071	666	memb
1083	1	isfinset
1093	18804	applyprim
1198	23613	apply
1239	28	dcprim
1261	1	readfname

Factorial function

Purpose Computes the factorial of n, where n = 0, 1, 2, ...

Discussion

Factorial functions written in ELC and Pascal have been included to compare the relative efficiency of the interpreter versus a conventional high level language compiler. Factorial is computed for n = 1 to 10. The results are not suprising in that the Pascal version is much faster.

Source Code (ELC)

```
<letrec fact
   <lambda <n>
      <if <<call <var equal> <var n> <con 0>>
          <con 1>
          <call <var prod>
                <var n>
                <call <var fact>
                      <call <var subt> <var n> <con 1>>>
                                                        >>>>
 <call <var fact> <con 10>>
```

```
Results of ELC factorial function for n = 1..10
fact(0)
Enter Expression
    1
Evaluation Completed
```

```
*****************************************************************
                          Statistics
System time was            83 milliseconds
User time was             233 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
```

```
|dcprim                    84      |
|cons                     202      |
|readiden                  30      |
|readint                    4      |
|letrec                     6      |
|equal                      1      |
--------------------------------
  Total cells             327

fact(1)
Enter Expression
    1
Evaluation Completed

************************************************************
                        Statistics
System time was           83 milliseconds
User time was            266 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                    84   |
|cons                     212   |
|readiden                  30   |
|readint                    4   |
|letrec                     6   |
|equal                      2   |
|subt                       1   |
|prod                       1   |
--------------------------------
  Total cells             340

fact(2)
Enter Expression
    2
Evaluation Completed
```

```
**********************************************************
                        Statistics
System time was          133 milliseconds
User time was            233 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                  84    |
|cons                   222    |
|readiden                30    |
|readint                  4    |
|letrec                   6    |
|equal                    3    |
|subt                     2    |
|prod                     2    |
--------------------------------
  Total cells            353

fact(3)
Enter Expression
    6
Evaluation Completed

**********************************************************
                        Statistics
System time was          133 milliseconds
User time was            266 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                  34    |
|cons                   232    |
|readiden                30    |
|readint                  4    |
```

```
|letrec                    6      |
|equal                     4      |
|subt                      3      |
|prod                      3      |
--------------------------------
  Total cells              366

fact(4)
Enter Expression
   24
Evaluation Completed

****************************************************************
                        Statistics
System time was          216 milliseconds
User time was            283 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                  84     |
|cons                   242     |
|readiden                30     |
|readint                  4     |
|letrec                   6     |
|equal                    5     |
|subt                     4     |
|prod                     4     |
--------------------------------
  Total cells              379

fact(5)
Enter Expression
  120
Evaluation Completed

****************************************************************
```

```
                        Statistics
System time was         116 milliseconds
User time was           350 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                  84     |
|cons                   252     |
|readiden                30     |
|readint                  4     |
|letrec                   6     |
|equal                    6     |
|subt                     5     |
|prod                     5     |
--------------------------------
  Total cells           392

fact(6)
Enter Expression
  720
Evaluation Completed
```

****

```
                        Statistics
System time was         116 milliseconds
User time was           416 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                  34     |
|cons                   262     |
|readiden                30     |
|readint                  4     |
|letrec                   6     |
|equal                    7     |
```

```
|subt                          6      |
|prod                          6      |
--------------------------------------
  Total cells                405

fact(7)
Enter Expression
 5040
Evaluation Completed

**************************************************************
                          Statistics
System time was           133 milliseconds
User time was             400 milliseconds

--------------------------------------
|Module          Cells created       |
|------          -------------       |
|dcprim                  84          |
|cons                   272          |
|readiden                30          |
|readint                  4          |
|letrec                   6          |
|equal                    8          |
|subt                     7          |
|prod                     7          |
--------------------------------------
  Total cells             418

fact(8)
Enter Expression
40320
Evaluation Completed

**************************************************************
                          Statistics
System time was           133 milliseconds
```

```
User time was          433 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                  84    |
|cons                   282    |
|readiden                30    |
|readint                  4    |
|letrec                   6    |
|equal                    9    |
|subt                     8    |
|prod                     8    |
--------------------------------
 Total cells           431

fact(9)
Enter Expression
362880
Evaluation Completed

****************************************************************
                        Statistics
System time was        150 milliseconds
User time was          450 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                  84    |
|cons                   292    |
|readiden                30    |
|readint                  4    |
|letrec                   6    |
|equal                   10    |
|subt                     9    |
|prod                     9    |
```

```
---------------------------------
  Total cells            444

fact(10)
Enter Expression
3628800
Evaluation Completed

**************************************************************
                        Statistics
System time was         183 milliseconds
User time was           450 milliseconds

---------------------------------
|Module          Cells created  |
|-  ---           ----------    |
|dcprim                 84      |
|cons                  302      |
|readiden               30      |
|readint                 4      |
|letrec                  6      |
|equal                  11      |
|subt                   10      |
|prod                   10      |
---------------------------------
  Total cells            457
```

Factorial function written in Berkely Pascal

Source Code

```
program fact(input, output);
   var ans,n:integer;

   function factorial(n:integer):integer;
      var fact:integer;
      begin
         if n = 0 then
            fact := 1
         else
            fact := n * (factorial (n - 1));
      factorial := fact;
      end; {function factorial}

   begin
      writeln('Input n: ');
      readln(n);
      ans := factorial(n);
      writeln(ans);
      writeln('System Clock ',sysclock:10 ,' millisec');
      writeln('User Clock   ',clock:10 ,' millisec');
   end. {Program fact}
```

Results of factorial function in Berkeley Pascal, n= 1..10.

```
Input n=0
        1
System Clock          33 millisec
User Clock            16 millisec


Input n=1
        1
System Clock          33 millisec
User Clock             0 millisec
```

```
Input n=2
         2
System Clock          33 millisec
User Clock             0 millisec

Input n=3
         6
System Clock          33 millisec
User Clock             0 millisec

Input n=4
        24
System Clock          33 millisec
User Clock             0 millisec

Input n=5
       120
System Clock          33 millisec
User Clock             0 millisec

Input n=6
       720
System Clock          33 millisec
User Clock             0 millisec

Input n=7
      5040
System Clock          16 millisec
User Clock            16 millisec
```

```
Input n=8
    40320
System Clock          33 millisec
User Clock             0 millisec

Input n=9
   362880
System Clock          33 millisec
User Clock             0 millisec

Input n=10
  3628800
System Clock          66 millisec
User Clock            16 millisec
Profile for ELC factorial function
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984  test11.p
Profiled Thu Dec 13 09:56 1984

       Line       Count

          1           1           func
         33           1                printval
         36         401                cellcount
         62          11                equal
         63         954                nullp
         84        2064                first
         94        1132                rest
        122         302                cons
        183          10                subt
        208          10                prod
        273          11                equalp
        291          11                equal
        533         150                atomp
```

547	954	nullp
599	73	assoc
643	22	pairlis
663	1	printval
702	56	readval
704	78	nonblank
714	22	readlist
741	0	readrea
770	4	readint
827	129	digit
835	163	letter
843	30	readident
867	56	readval
879	11	evcon
880	115	evlis
881	42	apply
882	1	letrec
884	150	eval
947	1	letrec
1012	11	evcon
1041	115	evlis
1093	31	applyprim
1198	42	apply
1239	28	dcprim
1261	1	readfname

Fibonacci Sequence Generation Program(No 'let' statement)

Purpose

This program generates the first n elements of the fibonacci sequence.

Discussion

This function is educational in that it shows how efficiency of ELC programs can be improved through the use of the 'let' statement. The definition of the Fibonacci sequence is:

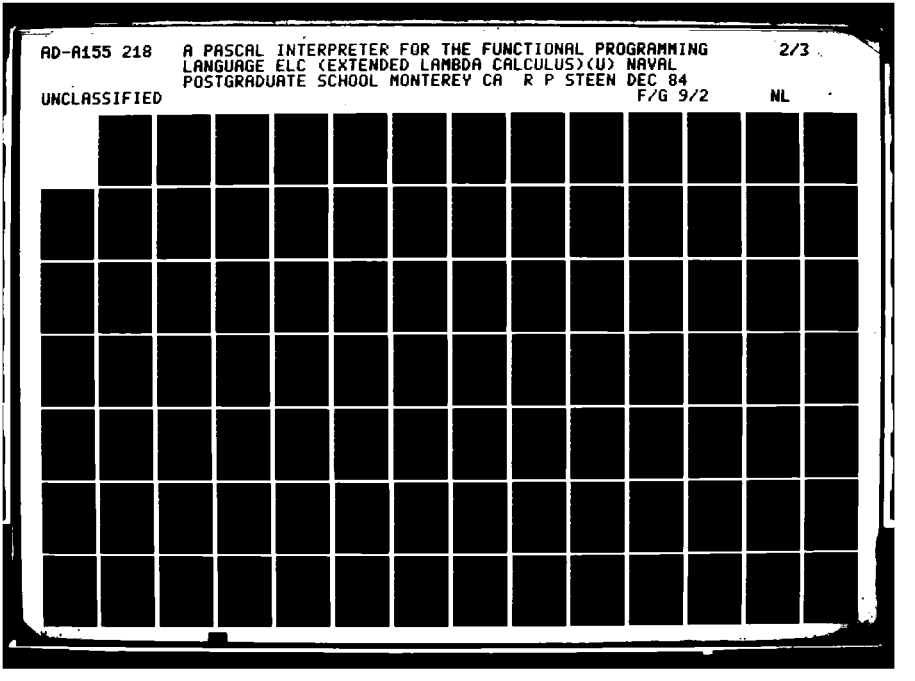
fib(1) = <1>
fib(2) = <1 1>
fib(n = 3, 4,...) =
  cons ((fib(n-1)sub 1) + (fib(n-2)sub 2)), fib(n-1)),
  where sub 1, 2 means subscript.

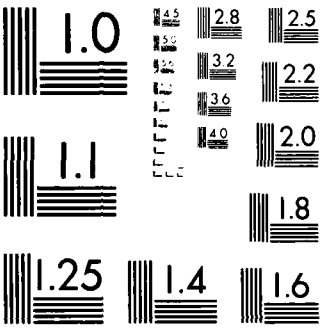
The time consuming part of this function, when written in ELC, is calculating fib of n-1 three times to find the next element of the sequence.This can be avoided by using a let statement to calculate fib(n-1) only once for each iteration. The system time taken to generate fib(10) when using the let statement was approximately .2 seconds compared to 13 seconds when a 'let' was not used.This in not suprising since when not using the 'let' the time of execution will increase exponentially as n increases.

Notice also that due to the nature of recursive construction of lists the sequence is constructed in reverse order. To correct this the reverse function is included and applied to the generated sequence before printing.

Source Code

```
<letrec reverse
   <lambda <L>
      <if <<call <var null> <var L>>
          <con <>>
```


AD-A155 218 A PASCAL INTERPRETER FOR THE FUNCTIONAL PROGRAMMING LANGUAGE ELC (EXTENDED LAMBDA CALCULUS)(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA R P STEEN DEC 84
2/3
UNCLASSIFIED
F/G 9/2
NL



```
          <call <var conr>
                <call <var reverse>
                      <call <var rest> <var L>>>
                <call <var sub> <var L> <con 1>> >>>>
<letrec fibo
  <lambda <n>
    <if <<call <var equal> <var n> <con 0>>
        <con <>>
        <if <<call <var equal> <var n> <con 1>>
            <con <1>>
            <if<<call <var equal> <var n> <con 2>>
                <ccn <1 1>>
                <call <var ccns>
                      <call <var sum>
                            <call <var sub>
                                  <call <var fibo>
                                        <call <var subt>
                                              <var n>
                                              <con 1>>>
                                  <con 1>>
                            <call <var sub>
                                  <call <var fibo>
                                        <call <var subt>
                                              <var n>
                                              <con 1>>>

                                  <con 2>>>
                      <call <var fibo>
                       <call <var subt> <var n> <con 1>>>>
                                                  >>>>>>>
<call <var reverse>
     <call <var fibo> <con 10>>>>>!

Results of fib(3..10) without a let statement
fib(3)
Enter Expression
```

```
<      1,      1,      2>

Evaluaticn Completed

****************************************************************
                        Statistics
System time was          183 milliseconds
User time was            733 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                  84     |
|cons                   428     |
|readiden               105     |
|readint                 11     |
|letrec                  12     |
|equal                   12     |
|subt                     3     |
|sum                      1     |
|null                     4     |
|conr                     2     |
--------------------------------
  Total cells            662

fib(4)
Enter Expression
<      1,      1,      2,      3>

Evaluation Completed

****************************************************************
                        Statistics
System time was          166 milliseconds
User time was           1250 milliseconds
```

```
--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                  84    |
|cons                   570    |
|readiden               105    |
|readint                 11    |
|letrec                  12    |
|equal                   39    |
|subt                    12    |
|sum                      4    |
|null                     5    |
|conr                     3    |
--------------------------------
  Total cells             845

fib(5)
Enter Expression
<      1,      1,      2,      3,      5>

Evaluation Completed

****************************************************************
                        Statistics
System time was          216 milliseconds
User time was           2566 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                  84    |
|cons                   976    |
|readiden               105    |
|readint                 11    |
|letrec                  12    |
|equal                  120    |
|subt                    39    |
```

```
|sum                          13      |
|null                          6      |
|conr                          4      |
--------------------------------------
  Total cells               1370

fib(6)
Enter Expression
<      1,       1,       2,       3,       5,       8>

Evaluation Completed

**************************************************************
                        Statistics
System time was          300 milliseconds
User time was           6350 milliseconds

--------------------------------------
|Module          Cells created       |
|------          -------------       |
|dcprim                  84          |
|cons                  2174          |
|readiden               105          |
|readint                 11          |
|letrec                  12          |
|equal                  363          |
|subt                   120          |
|sum                     40          |
|null                     7          |
|conr                     5          |
--------------------------------------
  Total cells               2921

fib(7)
Enter Expression
<      1,       1,       2,       3,       5,       8,      13>

Evaluation Completed
```

```
**********************************************************
                        Statistics
System time was          666 milliseconds
User time was          17766 milliseconds

---------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                   84    |
|cons                   5748    |
|readiden                105    |
|readint                  11    |
|letrec                   12    |
|equal                  1092    |
|subt                    363    |
|sum                     121    |
|null                      8    |
|conr                      6    |
---------------------------------
  Total cells            7550

fib(8)
Enter Expression
<     1,      1,      2,      3,      5,      8,     13,
     21>

Evaluation Completed

**********************************************************
                        Statistics
System time was          1633 milliseconds
User time was          51716 milliseconds

---------------------------------
|Module          Cells created  |
|------          -------------  |
```

```
|dcprim                    84      |
|cons                   16450      |
|readiden                 105      |
|readint                   11      |
|letrec                    12      |
|equal                   3279      |
|subt                    1092      |
|sum                      364      |
|null                       9      |
|conr                       7      |
--------------------------------
  Total cells            21413

fib(9)
Enter Expression
<      1,      1,      2,      3,      5,      8,     13,
      21,     34>

Evaluaticn Completed

****************************************************************
                        Statistics
System time was         4650 milliseconds
User time was         154450 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                    84   |
|cons                   48536   |
|readiden                 105   |
|readint                   11   |
|letrec                    12   |
|equal                   9840   |
|subt                    3279   |
|sum                     1093   |
|null                      10   |
```

```
|conr                    8     |
--------------------------------
  Total cells          62978

fib(10)
Enter Expression
<     1,     1,     2,     3,     5,
      8,    13,    21,    34,    55>

Evaluation Completed

****************************************************************
                        Statistics
System time was       13100 milliseconds
User time was        458983 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                 84      |
|cons                144774      |
|readiden              105      |
|readint                11      |
|letrec                 12      |
|equal                29523      |
|subt                  9840      |
|sum                   3280      |
|null                    11      |
|conr                     9      |
--------------------------------
  Total                  cells                        187649
****************************************************************
```

ELC Program to generate the Fibonacci Sequence (Using a let statement)

Source Code

```
<letrec reverse
   <lambda <L>
      <if <<call <var null> <var L>>
           <con <>>
           <call <var conr>
                 <call <var reverse>
                       <call <var rest> <var L>>>
                 <call <var sub> <var L> <con 1>>>>>>
 <letrec fibo
   <lambda <n>
     <if <<call <var equal> <var n> <con 0>>
          <con <>>
          <if <<call <var equal> <var n> <con 1>>
               <con <1>>
               <if<<call <var equal> <var n> <con 2>>
                   <con <1 1>>
                   <let <<f> <<call <var fibo>
                                    <call <var subt>
                                          <var n>
                                          <con 1>>>>
                             <call <var cons>
                                    <call <var sum>
                                          <call <var first>
                                                <var f>>
                                          <call <var first>
                                             <call<var rest>
                                                  <var f>>>>
                                    <var f>>>>>>>>>>>
 <call <var reverse>
      <call <var fibo> <con 10>>>>>!
```

```
 Results of fibonacci generating function with 'let'
statement
 fib (3..10)

 fib(3)
 Enter Expression
 <      1,       1,       2>

 Evaluation Completed

*****************************************************************
                        Statistics
 System time was          133 milliseconds
 User time was            700 milliseconds

 -------------------------------
 |Module          Cells created  |
 |------          -------------  |
 |dcprim                  84     |
 |cons                   392     |
 |readiden                95     |
 |readint                  9     |
 |letrec                  12     |
 |equal                    6     |
 |subt                     1     |
 |sum                      1     |
 |null                     4     |
 |conr                     2     |
 -------------------------------
   Total cells            606

 fib(4)
 Enter Expression
 <      1,       1,       2,       3>

 Evaluation Completed

*****************************************************************
```

```
                        Statistics
System time was          183 milliseconds
User time was            783 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                  84     |
|cons                   427     |
|readiden                95     |
|readint                  9     |
|letrec                  12     |
|equal                    9     |
|subt                     2     |
|sum                      2     |
|null                     5     |
|conr                     3     |
--------------------------------
  Total cells            648

fib(5)
Enter Expression
<      1,      1,      2,      3,      5>

Evaluation Completed

************************************************************
                        Statistics
System time was          166 milliseconds
User time was            866 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                  84     |
|cons                   462     |
|readiden                95     |
```

```
|readint                    9      |
|letrec                    12      |
|equal                     12      |
|subt                       3      |
|sum                        3      |
|null                       6      |
|conr                       4      |
--------------------------------
  Total cells             690

fib(6)
Enter Expression
<      1,       1,       2,       3,       5,       8>

Evaluation Completed

****************************************************************
                        Statistics
System time was          133 milliseconds
User time was           1066 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                    84   |
|cons                     497   |
|readiden                  95   |
|readint                    9   |
|letrec                    12   |
|equal                     15   |
|subt                       4   |
|sum                        4   |
|null                       7   |
|conr                       5   |
--------------------------------
  Total cells             732
```

```
fib(7)
Enter Expression
<      1,       1,       2,       3,       5,       8,      13>

Evaluation Completed

************************************************************
                        Statistics
System time was          183 milliseconds
User time was           1100 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                   84    |
|cons                    532    |
|readiden                 95    |
|readint                   9    |
|letrec                   12    |
|equal                    18    |
|subt                      5    |
|sum                       5    |
|null                      8    |
|conr                      6    |
--------------------------------
  Total cells             774

fib(8)
Enter Expression
      1,       1,       2,       3,       5,       8,      13,
      1>

Evaluation Completed

************************************************************
                        Statistics
System time was          183 milliseconds
User time was           1216 milliseconds
```

```
|len                          3     |
--------------------------------
  Total cells               1218


Results of restriction (interpreted)
Enter Expression
<<      6,      5>
, <      8,      9>
>

Evaluation Completed

****************************************************************
                        Statistics
System time was          516 milliseconds
User time was          17966 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                   84    |
|cons                    837    |
|readiden                220    |
|readint                   8    |
|letrec                   24    |
|eval                      1    |
|equal                    12    |
|null                     29    |
|len                       3    |
--------------------------------
  Total cells               1218


Profile for restriction program
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984  test11.p
```

```
                                    <call <var first>
                                          <var T>>>>
                          <call <var rest> <var T>>
                          <call <var cons>
                                <call <var first> <var T>>
                                <call <var restric>
                                      <call <var rest> <var T>>
                                      <var x>>>
               <con Tnotffunc>>>>>>>>
     <call <var restric>
          <call <var repr> <finset <3 4> <6 5> <8 9>> >
          <con 3> >>>>>>>!

     Results of restriction function (Compiled)
     Enter Expression
     <<     6,      5>
     , <     8,      9>
     >

     Evaluation Completed

*****************************************************************
                          Statistics
     System time was        216 milliseconds
     User time was         1966 milliseconds

     --------------------------------
     |Module          Cells created  |
     |------          -------------  |
     |dcprim                  84     |
     |cons                   837     |
     |readiden               220     |
     |readint                  8     |
     |letrec                  24     |
     |eval                     1     |
     |equal                   12     |
     |null                    29     |
```

```
            <con true>
            <if <<call <var equal>
                       <call <var len>
                             <call <var first> <var T>>>
                       <con 2>>
                  <if <<call <var equal>
                             <call <var member>
                                   <call <var first>
                                         <call <var firstlist>
                                         <var T>>>
                                   <call <var rest>
                                         <call <var firstlist>
                                         <var T>>>>
                             <con true>>
                        <con false>
                        <call <var isfinfunc>
                              <call <var rest>
                                    <var T>>>>>
                  <con false>>>>>>
<let <<repr> <<lambda <T>
                  <if <<call <var equal>
                             <call <var first> <var T>>
                             <con finset>>
                        <call <var rest> <var T>>
                        <con nofinfnc>>>>>
<letrec restric
  <lambda <T k>
     <if <<call <var equal>
                <call <var isfinfunc> <var T>>
                <con true>>
          <if <<call <var null> <var T>>
              <con <>>
              <if <<call <var equal>
                         <var k>
                         <call <var first>
```

Restriction

Purpose

Restriction takes a finite function, T, (table of attribute value pairs), and returns a finite function exactly like T except that one of the pairs has been removed. If the pair to be deleted is not a member of the finite function then T is returned (this is tolerant evaluation).

Practical Application Restriction could be used to delete records from a database.

Source Code

```
<letrec member
   <lambda <x L>
      <if <<call <var null> <var L>>
          <con false>
          <if <<call <var equal>
                     <var x>
                     <call <var first> <var L>>>
              <con true>
              <call <var member>
                    <var x>
                    <call <var rest> <var L>>>>>>>>
 <letrec firstlist
   <lambda <L>
      <if <<call <var null> <var L>>
          <con <>>
          <call <var cons>
                <call <var first>
                      <call <var first> <var L>>>
                <call <var firstlist>
                      <call <var rest> <var L>>>>>>>
 <letrec isfinfunc
   <lambda <T>
      <if <<call <var null> <var T>>
```

880	606	evlis
881	202	apply
882	1	letrec
884	761	eval
947	1	letrec
1012	51	evcon
1041	606	evlis
1093	151	applyprim
1198	202	apply
1239	28	dcprim
1261	1	readfname

```
|sum                          50      |
--------------------------------------

  Total cells               1192

Profile for interval function m, n 1-50
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984  test11.p
Profiled Thu Dec 13 13:54 1984

       Line        Count

          1            1            func
         33           51                 printval
         36         1136                 cellcount
         63         5487                 nullp
         84        11793                 first
         94         6357                 rest
        122          963                 cons
        158           50                 sum
        330           51                 GTp
        340           51                 GT
        539          761                 atomp
        547         5487                 nullp
        599          454                 assoc
        643          153                 pairlis
        663           51                 printval
        702           63                 readval
        704           88                 nonblank
        714           25                 readlist
        770            3                 readint
        827          151                 digit
        835          189                 letter
        843           35                 readident
        867           63                 readval
        879           51                 evcon
```

```
Evaluation Completed

**********************************************************
                        Statistics
System time was          266 milliseconds
User time was           1716 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                   84   |
|cons                    963   |
|readiden                 35   |
|readint                   3   |
|letrec                    6   |
|GT                       51   |
|sum                      50   |
--------------------------------
  Total cells           1192


Results of interval (interpreted)

**********************************************************
                        Statistics
System time was          616 milliseconds
User time was          19966 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                   84   |
|cons                    963   |
|readiden                 35   |
|readint                   3   |
|letrec                    6   |
|GT                       51   |
```

Interval Generating Function

Purpose

Generates a sequence of natural numbers from m to n, where m,n are two natural numbers and m < n.

Practical Application

Interval is very useful when generating tables of information. In the next example interval is used to generate a table of trigonometric values for all angles between 0 and 90 degrees.

Source Code

```
<letrec interval
   <lambda <m n>
      <if <<call <var GT> <var m> <var n>>
            <con <>>
            <call <var cons>
                  <var m>
                  <call <var interval>
                              <call <var sum>
                                    <var m> <con 1>>
                              <var n>> >>>>
 <call <var interval> <con 1> <con 50>>>!
```

Results of interval generation program (m = 1, n = 50, Compiled)

```
 Enter Expression
<     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>
```

539	466	atomp
547	3862	nullp
564	11	null
599	226	assoc
643	56	pairlis
663	11	printval
702	180	readval
704	256	nonblank
714	76	readlist
770	9	readint
827	410	digit
835	514	letter
843	95	readident
867	180	readval
879	38	evcon
880	375	evlis
881	136	apply
882	2	letrec
884	466	eval
947	2	letrec
1012	38	evcon
1041	375	evlis
1093	116	applyprim
1198	136	apply
1239	28	dcprim
1261	1	readfname

```
Use:  ime              0 millisec

fib(5)
Input n:
        1          1          2          3          5
        8         13         21         34
System time          33 millisec
User time             0 millisec

fib(10)
Input n:
        1          1          2          3          5
        8         13         21         34         55
System time          33 millisec
User time             0 millisec

Profile for ELC fibonacci sequence generating functions
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984  test11.p
Profiled Thu Dec 13 09:41 1984

      Line       Count

         1           1           func
        33          11               printval
        36         844               cellcount
        62          27               equal
        63        3862               nullp
        84        8188               first
        94        4372               rest
       122         637               cons
       137          10               conr
       158           8               sum
       183           8               subt
       258          10               sub
       273          27               equalp
       291          27               equal
```

```
fib(3)
Input n:
        1          1          2
System time         33 millisec
User time            0 millisec

fib(4)
Input n:
         1          1          2          3
System time         50 millisec
User time            0 millisec

fib(5)
Input n:
        1          1          2          3          5
System time         33 millisec
User time            0 millisec

fib(6)
Input n:
        1          1          2          3          5
        8
System time         33 millisec
User time            0 millisec

fib(7)
Input n:
        1          1          2          3          5
        8         13
System time         33 millisec
User time            0 millisec

fib(8)
Input n:
        1          1          2          3          5
        8         13         21
System time         33 millisec
```

Pascal Source Code for Fibonacci Sequence Generator

```
program fib(input, output);
     const max =100;
     type seq = 1..max  of integer;
     var fibseq: seq;
         n,c: integer;

  procedure fib(n,i:integer);
     begin
     if i <= n then begin
        if (i = 1) or (i = 2) then begin
           fibseq(.i.) := 1;
           fib(n,i + 1);
           end
        else if i >= 3 then begin
           fibseq(.i.) := fibseq(.i-1.) + fibseq(.i-2.);
           fib(n,i + 1);
           end
        end;
     end; {procedure fib}

  begin
     writeln('Input n: ');
     read(n);
     fib(n,1);
     for c := 1 to max do begin
        if fibseq(.c.) <> 0 then
           write (fibseq(.c.));
        end;
     writeln;
     writeln('System time',sysclock:10,' millisec');
     writeln('User time  ',clock:10,' millisec');
  end. {Program fib}
```

Results of fibonacci sequence generator in Pascal

```
|equal                    24      |
|subt                      7      |
|sum                       7      |
|null                     10      |
|conr                      8      |
--------------------------------
  Total cells             858

fib(10
Enter Expression
<      1,      1,      2,      3,      5,
       8,     13,     21,     34,     55>


Evaluation Completed

****************************************************************
                        Statistics
System time was          166 milliseconds
User time was           1466 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                   84    |
|cons                    637    |
|readiden                 95    |
|readint                   9    |
|letrec                   12    |
|equal                    27    |
|subt                      8    |
|sum                       8    |
|null                     11    |
|conr                      9    |
--------------------------------
  Total cells             900
```

Module	Cells created
dcprim	84
cons	567
readiden	95
readint	9
letrec	12
equal	21
subt	6
sum	6
null	9
conr	7
Total cells	816

```
fib(9)
Enter Expression
<     1,     1,     2,     3,     5,     8,     13,
     21,    34>

Evaluation Completed
```

****

Statistics

System time was 200 milliseconds
User time was 1300 milliseconds

Module	Cells created
dcprim	84
cons	602
readiden	95
readint	9
letrec	12

Profiled Thu Dec 13 14:50 1984

Line	Count	func
1	1	
33	7	printval
36	1162	cellcount
62	12	equal
63	4184	nullp
84	8859	first
94	4699	rest
122	837	cons
273	12	equalp
291	12	equal
539	441	atomp
547	4184	nullp
564	29	null
576	9	lenp
587	3	len
599	228	assoc
643	69	pairlis
663	7	printval
702	387	readval
704	546	nonblank
714	159	readlist
770	8	readint
827	1011	digit
835	1239	letter
843	220	readident
879	41	evcon
880	325	evlis
881	145	apply
882	4	letrec
884	441	eval
947	4	letrec
1012	41	evcon

1041	325	evlis
1093	115	applyprim
1198	145	apply
1239	28	dcprim
1261	1	readfname

Vectorproduct Function

Purpose

Vectorproduct returns the pairwise products of two lists of numbers.

Practical Application

This function could be used to calculate the state tax owed by military employees, since different states have different rates of taxation. One vector would be the list of salaries and the other the rates of taxation.

Source Code

```
<letrec map
   <lambda <f>
      <lambda <L>
         <if <<call <var null> <var L>>
               <con <>>
               <call <var cons>
                   <call <var f> <call <var first><var L>>>
                   <call <call <var map> <var f>>
                           <call <var rest> <var L>>>>>>>>
 <letrec prodlist
   <lambda <L>
      <if <<call <var null> <var L>>
           <con 1>
           <call <var prod>
                 <call <var sub> <var L> <con 1>>
                 <call <var prodlist>
                       <call <var rest> <var L>>>>>>>
 <letrec pairlist
   <lambda <L M>
      <if <<call <var equal>
                 <call <var len> <var L>>
                 <call <var len> <var M>>>
           <if <<call <var null> <var L>>
```

```
                <if <<call <var null> <var M>>
                     <con <>>
                     <con <>>>>
                <call <var cons>
                      <call <var cons>
                            <call <var first> <var L>>
                            <call <var cons>
                                  <call <var first> <var M>>
                                  <con <>> >>
                      <call <var pairlist>
                            <call <var rest> <var L>>
                            <call <var rest> <var M>>>>>>
          <con errorpl> >>>
 <call
  <call <var map> <var prodlist>>
      <call <var pairlist>
            <list <con 5> <con 8> <con 4> <con 9>>
            <list <con 2> <con 20> <con 7> <con 3>>>>>>>!

 Results of vectorprod function (Compiled)
 Enter Expression
 <    10,    160,     28,     27>

 Evaluation Completed

**************************************************************
                        Statistics
 System time was         216 milliseconds
 User time was          1583 milliseconds

 --------------------------------
 |Module          Cells created  |
 |------          -------------  |
 |dcprim                  84     |
 |cons                   720     |
 |readiden               150     |
 |readint                 10     |
```

```
|letrec                    18      |
|len                       10      |
|equal                      5      |
|null                      23      |
|eval                       5      |
|prod                       8      |
--------------------------------
  Total cells             1033
```

```
Results of vectorproduct function (Interpreted)
Enter Expression
<    10,    160,     28,     27>

Evaluation Completed
```

```
*****************************************************************
                          Statistics
System time was             483 milliseconds
User time was             15066 milliseconds
```

```
--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                   84    |
|cons                    720    |
|readiden                150    |
|readint                  10    |
|letrec                   18    |
|eval                      5    |
|len                      10    |
|equal                     5    |
|null                     23    |
|prod                      8    |
--------------------------------
  Total cells             1033
```

Profile for vectorproduct function
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984 test11.p
Profiled Thu Dec 13 15:09 1984

line	Count	
1	1	func
33	5	printval
36	977	cellcount
62	5	equal
63	3579	nullp
84	7535	first
94	4038	rest
122	720	cons
208	8	prod
258	8	sub
273	5	equalp
291	5	equal
539	395	atomp
547	3579	nullp
564	23	null
576	30	lenp
587	10	len
599	202	assoc
643	59	pairlis
663	5	printval
702	272	readval
704	384	nonblank
714	112	readlist
770	10	readint
827	659	digit
835	819	letter
843	150	readident

867	272	readval
879	28	evcon
880	310	evlis
881	129	apply
882	3	letrec
884	395	eval
947	3	letrec
1012	28	evcon
1041	310	evlis
1093	102	applyprim
1198	129	apply
1239	28	dcprim
1261	1	readfname

Filter Function

Purpose

Filter allows the user to extract information from a list based on a Boolean condition. In the example given, all numbers greater than 2000 are extracted from the list.

Practical Application

Filter is another function that could be useful when dealing with databases. Users of relational database systems use filtering every time they write a query. Imagine that the elements of the example are salaries. The query demonstrated is to find all salaries greater than 2000.

Source Code

```
<letrec fil
  <lambda <bool arg>
     <lambda <L>
        <if <<call <var null> <var L>>
            <con <>>
            <if <<call <var bcol>
                       <call <var sub> <var L>
                       <con 1>>
                 <var arg>>
                 <call <var cons>
                       <call <var sub> <var L> <con 1>>
                       <call <call <var fil>
                                   <var bool>
                                   <var arg>>
                             <call <var rest> <var L>>>>
                 <call <call <var fil>
                             <var bool>
                             <var arg>>
                       <call <var rest> <var L>>>>>>>>>
<call
```

```
   <call <var fil> <var GE> <ccn 2000>>
<list <con 1000> <ccn 12000> <con 2005> <con 3400>
      <con 3305> <con 134> <con 2001> <con 3500>
      <con 2209> <ccn 1999>>>>!

Results of filter function(Interpreted)
Enter Expression
< 12000,   2005,   3400,   3305,   2001,   3500,   2209>

Evaluation Completed

*****************************************************************
                        Statistics
System time was          416 milliseconds
User time was           9083 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                  84    |
|cons                   562    |
|readiden                79    |
|readint                 13    |
|letrec                   6    |
|eval                    11    |
|null                    11    |
|GE                      10    |
--------------------------------
  Total cells             776


Result of filter function (Compiled)
Enter Expression
< 12000,   2005,   3400,   3305,   2001,   3500,   2209>

Evaluation Completed

*****************************************************************
```

Statistics

System time was 216 milliseconds
User time was 816 milliseconds

Module	Cells created
dcprim	84
cons	562
readiden	79
readint	13
letrec	6
eval	11
null	11
GE	10
Total cells	776

Profile for the filtering function
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984 test11.p
Profiled Thu Dec 13 13:40 1984

Line	Count	
1	1	func
33	8	printval
36	720	cellcount
63	1838	nullp
84	4010	first
94	2185	rest
122	562	cons
258	17	sub
357	10	GEp
367	10	GE
539	275	atomp

547	1838	nullp
564	11	null
599	135	assoc
643	55	pairlis
663	8	printval
702	151	readval
704	210	nonblank
714	59	readlist
770	13	readint
827	332	digit
835	424	letter
843	79	readident
867	151	readval
879	21	evcon
880	210	evlis
881	77	apply
882	1	letrec
884	275	eval
947	1	letrec
1012	21	evcon
1041	210	evlis
1093	55	applyprim
1198	77	apply
1239	28	dcprim
1261	1	readfname

Periodic Sequence Generator

Purpose

This program simply illustrates the interpreter's ability to generate a recursive sequence.

Source Code

```
<letrec reverse
  <lambda <L>
     <if <<call <var null> <var L>>
          <con <>>
          <call <var conr>
                <call <var reverse>
                      <call <var rest> <var L>>>
                <call <var sub> <var L> <con 1>>>>>>
<letrec fibo
  <lambda <n>
         <if <<call <var equal> <var n> <con 1>>
              <con <2>>
              <if<<call <var equal> <var n> <con 2>>
                  <ccn <9 2>>
                  <let <<f> <<call <var fibo>
                                   <call <var subt>
                                         <var n>
                                         <con 1>>>>
                            <call <var cons>
                                  <call <var subt>
                                        <call <var first>
                                              <var f>>
                                        <call <var first>
                                           <call <var rest>
                                                 <var f>>>>
                                  <var f>>>>>>>>>
<call <var reverse>
     <call <var fibo> <con 24>>>>>!
```

```
Results of generating the first 24 elements of a periodic
sequence, where x1 = 2, x2 = 9, and xk = (xk-1)-(xk-2)
for k =3, 4, 5, ...

Enter Expression
<     2,      9,      7,     -2,     -9,     -7,
      2,      9,      7,     -2,     -9,     -7,
      2,      9,      7,     -2,     -9,     -7,
      2,      9,      7,     -2,     -9,     -7>

Evaluation Completed

************************************************************

                        Statistics
System time was          283 milliseconds
User time was           2916 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                  84    |
|cons                  1064    |
|readiden                87    |
|readint                  8    |
|letrec                  12    |
|equal                   46    |
|subt                    44    |
|null                    25    |
|conr                    23    |
--------------------------------
  Total cells            1393


Profile of seq2 (periodic function)
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984  test11.p
Profiled Thu Dec 13 10:10 1984
```

Line	Count	
1	1	func
33	21	printval
36	1181	cellcount
62	38	equal
63	7652	nullp
84	16163	first
94	8626	rest
122	932	cons
137	20	conr
183	36	subt
258	20	sub
273	38	equalp
291	38	equal
539	881	atomp
547	7652	nullp
564	21	null
593	438	assoc
643	116	pairlis
663	21	printval
702	163	readval
704	231	nonblank
714	68	readlist
770	8	readint
827	379	digit
835	474	letter
843	87	readident
867	163	readval
879	59	evcon
880	738	evlis
881	267	apply
882	2	letrec
884	881	eval
947	2	letrec

```
, <    44, 0.694658, 0.719340, 0.965689>
, <    45, 0.707107, 0.707107, 1.000000>
, <    46, 0.719340, 0.694658, 1.035530>
, <    47, 0.731354, 0.681998, 1.072369>
, <    48, 0.743145, 0.669131, 1.110613>
, <    49, 0.754710, 0.656059, 1.150368>
, <    50, 0.766044, 0.642788, 1.191754>
, <    51, 0.777146, 0.629320, 1.234897>
, <    52, 0.788011, 0.615661, 1.279942>
, <    53, 0.798636, 0.601815, 1.327045>
, <    54, 0.809017, 0.587785, 1.376382>
, <    55, 0.819152, 0.573576, 1.428148>
, <    56, 0.829038, 0.559193, 1.482561>
, <    57, 0.838671, 0.544639, 1.539865>
, <    58, 0.848048, 0.529919, 1.600335>
, <    59, 0.857167, 0.515038, 1.664279>
, <    60, 0.866025, 0.500000, 1.732051>
, <    61, 0.874620, 0.484810, 1.804048>
, <    62, 0.882948, 0.469472, 1.880726>
, <    63, 0.891007, 0.453990, 1.962611>
, <    64, 0.898794, 0.438371, 2.050304>
, <    65, 0.906308, 0.422618, 2.144507>
, <    66, 0.913545, 0.406737, 2.246037>
, <    67, 0.920505, 0.390731, 2.355852>
, <    68, 0.927184, 0.374607, 2.475087>
, <    69, 0.933580, 0.358368, 2.605089>
, <    70, 0.939693, 0.342020, 2.747477>
, <    71, 0.945519, 0.325568, 2.904211>
, <    72, 0.951057, 0.309017, 3.077684>
, <    73, 0.956305, 0.292372, 3.270853>
, <    74, 0.961262, 0.275637, 3.487414>
, <    75, 0.965926, 0.258819, 3.732051>
, <    76, 0.970296, 0.241922, 4.010781>
, <    77, 0.974370, 0.224951, 4.331476>
, <    78, 0.978148, 0.207912, 4.704630>
```

```
, <     9, 0.156434, 0.987688, 0.158384>
, <    10, 0.173648, 0.984808, 0.176327>
, <    11, 0.190809, 0.981627, 0.194380>
, <    12, 0.207912, 0.978148, 0.212557>
, <    13, 0.224951, 0.974370, 0.230868>
, <    14, 0.241922, 0.970296, 0.249328>
, <    15, 0.258819, 0.965926, 0.267949>
, <    16, 0.275637, 0.961262, 0.286745>
, <    17, 0.292372, 0.956305, 0.305731>
, <    18, 0.309017, 0.951057, 0.324920>
, <    19, 0.325568, 0.945519, 0.344328>
, <    20, 0.342020, 0.939693, 0.363970>
, <    21, 0.358368, 0.933580, 0.383864>
, <    22, 0.374607, 0.927184, 0.404026>
, <    23, 0.390731, 0.920505, 0.424475>
, <    24, 0.406737, 0.913545, 0.445229>
, <    25, 0.422618, 0.906308, 0.466308>
, <    26, 0.438371, 0.898794, 0.487733>
, <    27, 0.453990, 0.891007, 0.509525>
, <    28, 0.469472, 0.882948, 0.531709>
, <    29, 0.484810, 0.874620, 0.554309>
, <    30, 0.500000, 0.866025, 0.577350>
, <    31, 0.515038, 0.857167, 0.600861>
, <    32, 0.529919, 0.848048, 0.624869>
, <    33, 0.544639, 0.838671, 0.649408>
, <    34, 0.559193, 0.829038, 0.674509>
, <    35, 0.573576, 0.819152, 0.700208>
, <    36, 0.587785, 0.809017, 0.726543>
, <    37, 0.601815, 0.798636, 0.753554>
, <    38, 0.615661, 0.788011, 0.781286>
, <    39, 0.629320, 0.777146, 0.809784>
, <    40, 0.642788, 0.766044, 0.839100>
, <    41, 0.656059, 0.754710, 0.869287>
, <    42, 0.669131, 0.743145, 0.900404>
, <    43, 0.681998, 0.731354, 0.932515>
```

```
                    <var x>>
                    <call
                      <call <var pam>
                            <call <var rest><var F>>>
                            <var x>>>>>>>
<letrec interval
  <lambda <m n>
     <if <<call <var GT> <var m> <var n>>
           <con <>>
           <call <var cons>
                  <var m>
                  <call <var interval>
                            <call <var sum>
                                  <var m>
                                  <con 1>>
                            <var n>> >>>>
<call
  <call <var map>
  <call <var pam> <list <var id> <var sin>
                        <var cos> <var tan>>>>
<call <var interval> <con 0> <con 90>>>>>>!
```

Results of mapping the pam function across a list to generate the table of trigonometric values for angles 0 - 90 degrees.

```
Enter Expression
<<      0, 0.000000, 1.000000, 0.000000>
, <     1, 0.017452, 0.999848, 0.017455>
, <     2, 0.034899, 0.999391, 0.034921>
, <     3, 0.052336, 0.998630, 0.052408>
, <     4, 0.069756, 0.997564, 0.069927>
, <     5, 0.087156, 0.996195, 0.087489>
, <     6, 0.104528, 0.994522, 0.105104>
, <     7, 0.121869, 0.992546, 0.122785>
, <     8, 0.139173, 0.990268, 0.140541>
```

Trig Table Generating Program

Purpose

Generates a table of trigonometric values for all angles in the interval 0 to 90 degrees.

Discussion

This program demonstrates the value of the interval function combined with the map functional. The reverse of the map functional, (pam), is also used. Map takes one function and applies it to all the elements of a list, where pam takes a list of functions and applies each one to the same argument. It is clear that mapping the pam function across the interval 0 to 90 produces the desired results. This program also illustrates the value of the 'id' primitive which allows the first element of each of the sublists in the result to be the angle.

Source Code

```
<letrec map
   <lambda <f>
      <lambda <L>
         <if <<call <var null> <var L>>
              <con <>>
              <call <var cons>
                  <call <var f> <call <var first><var L>>>
                  <call <call <var map> <var f>>
                         <call <var rest> <var L>>>>>>>>
 <letrec pam
   <lambda <F>
      <lambda <x>
         <if <<call <var null> <var F>>
              <con <>>
              <call <var cons>
                    <call
                      <call <var first> <var F>>
```

947	1	letrec
1012	6	evcon
1041	169	evlis
1093	45	applyprim
1198	52	apply
1239	28	dcprim
1261	1	readfname

Profiled Thu Dec 13 15:03 1984

Line	Count	func
1	1	
33	13	printval
36	614	cellcount
62	6	equal
63	1656	nullp
84	3439	first
94	1861	rest
122	472	cons
183	5	subt
233	1	divi
273	6	equalp
291	6	equal
539	183	atomp
547	1656	nullp
576	11	lenp
587	1	len
599	86	assoc
643	32	pairlis
663	13	printval
702	156	readval
704	218	nonblank
714	62	readlist
770	3	readint
827	375	digit
835	469	letter
843	91	readident
867	156	readval
879	6	evcon
880	169	evlis
881	52	apply
882	1	letrec
884	183	eval

```
                <var L>>>>
        <call <var split> <list a b c d e f i o e t>>>>>!

 Results of using the split function  to divide a 10 element
list
 Enter Expression
 <<a, b, c, d, e>
 , <f, i, o, e, t>
 >

 Evaluation Completed

*****************************************************************
                          Statistics
 System time was          166 milliseconds
 User time was            833 milliseconds

 --------------------------------
 |Module          Cells created  |
 |------          -------------  |
 |dcprim                  84     |
 |cons                   472     |
 |readiden                91     |
 |readint                  3     |
 |letrec                   6     |
 |eval                     1     |
 |len                      1     |
 |divi                     1     |
 |equal                    6     |
 |subt                     5     |
 --------------------------------
   Total cells            670


 Profile of split (ten element list)
 Berkeley Pascal PXP -- Version 2.12 (5/11/83)
 Wed Dec 12 12:48 1984  test11.p
```

Split Function

Purpose

Split takes a list and divides it into two equal size lists.

Practical Application

The split function illustrates how functional languages lend themselves to parallel computer operations. If quick-sort was implemented using split then once the list was initially separated into two lists, two processors could work on those two lists, etc..

Source Code

```
<letrec splitaux
  <lambda <k L>
     <if <<call <var equal> <var k> <con 0>>
          <call <var cons>
                <con <>>
                <call <var cons>
                      <var L>
                      <con <>>>>
          <let <<r> <<call <var splitaux>
                           <call <var subt> <var k> <con 1>>
                           <call <var rest> <var L>>>>
                <call <var cons>
                     <call <var cons>
                           <call <var first> <var L>>
                           <call <var first> <var r>>>
                     <call <var rest> <var r>>>>>>>>
<let <<split>
     <<lambda <L>
        <call <var splitaux>
              <call <var divi>
                    <call <var len> <var L>>
                    <con 2>>
```

122	907	cons
273	17	equalp
291	13	equal
539	711	atomp
547	6758	nullp
564	46	null
576	18	lenp
587	6	len
599	371	assoc
643	96	pairlis
663	13	printval
702	301	readval
704	423	nonblank
714	122	readlist
770	9	readint
827	794	digit
835	973	letter
843	170	readident
867	301	readval
879	65	evcon
880	532	evlis
881	241	apply
882	3	letrec
884	711	eval
947	3	letrec
1012	65	evcon
1041	532	evlis
1061	10	membp
1071	6	memb
1083	1	isfinset
1093	195	applyprim
1198	241	apply
1239	28	dcprim
1261	1	readfname

```
Evaluation Completed

****************************************************************

                        Statistics
System time was          633 milliseconds
User time was          24433 milliseconds

--------------------------------
|Module          Cells created |
|------          ------------- |
|dcprim                   84   |
|cons                    907   |
|readiden                170   |
|readint                   9   |
|letrec                   18   |
|null                     46   |
|len                       6   |
|equal                    13   |
|memb                      6   |
--------------------------------
  Total cells            1259


Profile for overlay program
Berkel  Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984  test11.p
Profiled Thu Dec 13 14:05 1984

      Line       Count

         1           1          func
        33          13               printval
        36        1203               cellcount
        62          13               equal
        63        6758               nullp
        84       14295               first
        94        7556               rest
```

```
<<      3,      4>
, <     6,      5>
, <     8,      9>
, <     7,      2>
>

Evaluation Completed
```

****

```
                        Statistics
System time was          183 milliseconds
User time was           2333 milliseconds
```

Module	Cells created
dcprim	84
cons	907
readiden	170
readint	9
letrec	18
null	46
len	6
equal	13
memb	6
Total cells	1259

```
Results of overlay (Interpreted)
Enter Expression
<<      3,      4>
, <     6,      5>
, <     8,      9>
, <     7,      2>
>
```

```
                    <call <var isfinfunc>
                          <call <var rest>
                                <var T>>>>>
               <con false>
                                                          >>>>>
<letrec overlay
  <lambda <T pr>
     <if <<call <var equal>
                <call <var isfinfunc> <var T>>
                <con true>>
          <if <<call <var null> <var T>>
              <call <var cons> <var pr> <con <>>>
              <if <<call <var equal>
                         <call <var first> <var pr>>
                         <call <var first>
                               <call <var first>
                                     <var T>>>>
                    <call <var overlay>
                          <call <var rest> <var T>>
                          <var pr>>
                    <call <var cons>
                          <call <var first> <var T>>
                          <call <var overlay>
                                <call <var rest> <var T>>
                                <var pr>>>
                                          >>>>
          <con Tnotffunc>
                                                  >>>
<call <var overlay>
     <call <var repr> <finset <3 4> <6 5> <8 9>>>
     <list 7 2> >>>>!
```

Results of overlay function adding the value <3 4>
to the table. (Compiled)

Enter Expression

Overlay Function

Purpose

Overlay takes a finite function, (table), and returns an identical table with an additional pair added.

Practical Application

Overlay could be used as a way to update a database.

Source Code

```
<letrec firstlist
   <lambda <L>
      <if <<call <var null> <var L>>
          <con <>>
          <call <var cons>
                <call <var first>
                      <call <var first> <var L>>>
                <call <var firstlist>
                      <call <var rest>
                            <var L>>>>>>>
 <letrec isfinfunc
   <lambda <T>
      <if <<call <var null> <var T>>
          <con true>
          <if <<call <var equal>
                     <call <var len>
                           <call <var first> <var T>>>
                     <con 2>>
              <if <<call <var memb>
                         <call <var first>
                               <call <var firstlist>
                                     <var T>>>
                         <call <var rest>
                               <call <var firstlist>
                                     <var T>>>>
                  <con false>
```

1012	59	evcon
1041	738	evlis
1093	227	applyprim
1198	267	apply
1239	28	dcprim
1261	1	readfname

```
, <     79, 0.981627, 0.190809, 5.144554>
, <     80, 0.984808, 0.173648, 5.671282>
, <     81, 0.987688, 0.156434, 6.313752>
, <     82, 0.990268, 0.139173, 7.115370>
, <     83, 0.992546, 0.121869, 8.144346>
, <     84, 0.994522, 0.104528, 9.514364>
, <     85, 0.996195, 0.087156, 11.430052>
, <     86, 0.997564, 0.069756, 14.300666>
, <     87, 0.998630, 0.052336, 19.081137>
, <     88, 0.999391, 0.034899, 28.636253>
, <     89, 0.999848, 0.017452, 57.289962>
, <     90, 1.000000, 'undef', 'undef'>
>

Evaluation Completed

****************************************************************

                        Statistics
System time was       4450 milliseconds
User time was       292650 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                  84     |
|cons                 10305     |
|readiden                121     |
|readint                   3     |
|letrec                   18     |
|eval                    457     |
|GT                       92     |
|sum                      91     |
|null                    547     |
|sinp                     91     |
|cosp                     91     |
|tanr                     91     |
```

```
--------------------------------
  Total cells          11991

Results of trig table generator (Compiled)
Evaluation Completed

*****************************************************************
                        Statistics
System time was        1216 milliseconds
User time was         24100 milliseconds

--------------------------------
|Module          Cells created  |
|------          -------------  |
|dcprim                   84    |
|cons                  10305    |
|readiden                121    |
|readint                   3    |
|letrec                   18    |
|GT                       92    |
|sum                      91    |
|eval                    457    |
|null                    547    |
|tanr                     91    |
|cosp                     91    |
|sinp                     91    |
--------------------------------
Total cells          11991


Profile for trigtable generating function
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984  test11.p
Profiled Thu Dec 13 09:36 1984

        Line        Count
```

1	1	func
33	456	printval
36	11935	cellcount
63	88505	nullp
84	189153	first
94	102572	rest
122	10305	cons
158	91	sum
330	92	GTp
340	92	GT
411	91	sinp
433	91	cosp
454	91	tann
539	10495	atomp
547	88505	nullp
564	547	null
599	5563	assoc
643	2284	pairlis
663	456	printval
702	211	readval
704	298	nonblank
714	87	readlist
770	3	readint
827	510	digit
835	634	letter
843	121	readident
867	211	readval
879	639	evcon
880	8118	evlis
881	3646	apply
882	3	letrec
884	10495	eval
947	3	letrec
1012	639	evcon

1041	8118	evlis
1093	2550	applyprim
1198	3646	apply
1239	28	dcprim
1261	1	readfname

Comparison of Programs Run With and Without the Memory Manager (MM)

General

The column labeled "left" in the following table refers to the number of cells that were in the freelist after evaluation of the program. This is caused by returning the cells that made the program list and is noteworthy because several programs could be loaded in the same file and evaluated without the danger of using all allocated memory.

	Cells Created				
Program	MM	Nc MM	Left	System TIME	User TIME
Reverse	381	401	46	416	9566
Revaux	427	427	46	533	1090
Append	531	561	51	566	10583
Map(sine)	599	629	95	533	12233
Halving	774	898	102	816	15800
Collate	718	748	72	700	17566
Factorial	427	457	18	416	7566
Interval	1042	1192	211	933	34483
Filter	755	776	147	516	14233
Periodic	1071	1393	8	1116	53416
Split	611	670	24	466	12300

```
(*Extended Lambda Calculus Interpreter*) program func(input, output);
  type
   filename = packed array (.1..80.) of char;
   tagtype = (lst, int, rea, alf, boo);
   list = @cell;
   (*Data structure for the cells of the program*)
   cell = record
            ref: integer;
            case tag: tagtype of
               lst: ( head, tail: list );
               int: ( ival: integer );
               rea: ( rval: real );
               alf: ( aval: alfa );
               boo: ( bval: boolean );
          end; (*Cell*)
   (*Data structure for free list*)
   freehdr = record
               numcells: integer;
               next: list;
             end;
   (*Data structure for tracking the number of cells created*)
```

```
    cntinfo = record
                modul: alfa;
                cellcnt: integer;
              end; (*cntinfo*)
    (*There are 26 modules that create cells in the interpreter.
     The array declared below is used to store information about
     the number of cells each module creates*)
    cntrec = array .)1..26.)  of cntinfo;
    var temp,primitives: list;
        ch: char;
        k,l: integer;
        infile: text;
        ans:char;
        callonly, diags,interac:boolean;
        dtmoyr, curtime: alfa;
        filearg: filename;
        newcells: integer;
        counts: cntrec;
        hdr: freehdr;

procedure printval(ls:list);  forward;

(*****************************************************************************
 * Function empty                                                            *
 *                                                                           *
```

```
 * Purpose: Checks the free list to see if there are any cells available *
 *          to be retrieved.                                              *
 *                                                                        *
 * Calls: None                                                            *
 *                                                                        *
 * Called by: sum, cons, subt, prod, divi, equal, LT, GT, GE, LE, sinp,   *
 *            cosp, tann, cot, sec, csc, atom, len, readrea, readint,     *
 *            readstring, readident, eval, letrec, memb                   *
 **************************************************************************)
function empty: boolean;
   begin
      empty := hdr.next = nil;
   end; (*Function empty*)

(**************************************************************************
 * Function freecell                                                      *
 *                                                                        *
 * Purpose: Retrieves cells from the freelist to be used where needed.    *
 *                                                                        *
 * Calls: None                                                            *
 *                                                                        *
 * Called by: cons, sum, subt, prod, divi, equal, LT, GT, GE, LE, sinp,   *
 *            cosp, tann, cot, sec, csc, atom, null, len, readrea,        *
 *            readint, readstring, readident, eval, letrec, memb          *
```

```
 ***************************************************************************)
function freecell: list;
   var C: list;
   begin
      C := hdr.next;
      hdr.next := C@.tail;
      hdr.numcells := hdr.numcells - 1;
      freecell := C;
   end; (*Function freecell*)

(***************************************************************************
 * Function  return                                                        *
 *                                                                         *
 * Purpose: Returns cells to the freelist when the cell's reference        *
 *          count becomes zero.                                            *
 *                                                                         *
 * Calls: None                                                             *
 *                                                                         *
 * Called by: decr                                                         *
 ***************************************************************************)
procedure return(C: list);
   begin
      if diags then begin
        writeln;
        writeln('*************************************************************' ;
```

```
        writeln('*Examining the cell being returned');
        writeln('*   the tag for C is-------> ',C@.tag);
        writeln('*   printing the contents of C:');
           printval(C);
        writeln;
        writeln('*****************************************************************');
        end;
      C@.tag := lst;
      C@.ref := 0;
      if empty then begin
         C@.tail := nil;
         hdr.next := C;
         hdr.numcells := hdr.numcells + 1;
         end
      else begin
         C@.tail := hdr.next;
         hdr.next := C;
         hdr.numcells := hdr.numcells + 1;
         end;
      writeln('In return number of cells in freelist is ',hdr.numcells);
   end; (*Procedure return*)

(*************************************************************************
 * Function decr                                                         *
 *                                                                       *
```

```
 * Purpose: Decrements the reference counts of cells that have references*
 *          destroyed because of overwriting pointers or because they    *
 *          become inaccessible.                                         *
 *                                                                       *
 * Calls: return                                                         *
 *                                                                       *
 * Called by:  ptrassn, initial, sub, equal, LT, assoc, pairlis,         *
 *             readlist, eval, letrec, evcon, applyprim, apply           *
 ************************************************************************)
procedure decr(C:list);
   begin
      if C = nil then
         (*do nothing*)
      else if C@.ref = 0 then
         return(C)
      else begin
         C@.ref := C@.ref - 1;
         if C@.ref = 0 then begin
            if C@.tag = lst then begin
               decr(C@.head);
               decr(C@.tail);
               return(C);
               end
            else if (C@.tag = alf)|(C@.tag = int)|
```

```
                    (C@.tag = rea)|(C@.tag = boo) then
                return(C)
            end
        end (*else begin*)
    end; (*Procedure decr*)

(*************************************************************************
 * Function ptrassn                                                      *
 *                                                                       *
 * Purpose: Automatically keeps track of the reference counts for two    *
 *          pointers, one of which overwrites the other.                 *
 *                                                                       *
 * Calls: decr                                                           *
 *                                                                       *
 * Called by: last, initial, cons, conr, assoc, readlist, eval, letrec,  *
 *            applyprim, apply                                           *
 *************************************************************************)
procedure ptrassn (var x: list; y: list);
    begin
        if diags then begin
            writeln('ref cnts of x and y in ptrassn if thy arent nil');
            if x <> nil then
              writeln(' x--> ',x@.ref);
            if y <> nil then
               writeln(' y--> ',y@.ref);
```

```
        end;
      if x <> nil then
        decr(x);
      if y <> nil then
        y@.ref := y@.ref + 1;
      x := y;
  end; (*Procedure ptrassn*)

(*************************************************************************
 * Function cellcount                                                    *
 *                                                                       *
 * Purpose: Tabulates the number of cells created by the functions in the*
 *          interpreter.                                                 *
 *                                                                       *
 * Calls:  None                                                          *
 *                                                                       *
 * Called by: conr, sum, subt, prod, divi, equal, LT, GT, GE, LE, sinp   *
 *            cosp, tann, cot, sec, csc, atom, null, len, readrea,       *
 *            readint, readstring, readident, eval, letrec, memb, dcprim *
 *************************************************************************)
procedure cellcount(i:integer; module:alfa);
  var m: integer;
      quit: boolean;
  begin
    m := 1;
```

```
        quit := false;
        newcells := newcells + i;
        repeat
           if counts(.m.).modul = 'empty' then begin
              with counts(.m.) do begin
                 modul := module;
                 cellcnt := cellcnt + i;
                 end;
              quit := true;
              end
           else if counts(.m.).modul = module then begin
              counts(.m.).cellcnt := counts(.m.).cellcount + i;
              quit := true;
              end;
           m := m + 1;
           if m = 27 then begin
              quit := true;
              end;
        until quit = true;
     end; (*Procedure cellcount*)

function equal(x,y:list):list;    forward;   function nullp (L:list):boolean;
forward;

(*************************************************************************
```

```
        end
    else
        P := freecell;
    with P@ do begin
        ref := 0;
        tag := int;
        ival := a@.ival * b@.ival;
        end;
    prod := P;
    end
else if (a@.tag = rea) and (b@.tag = rea) then begin
    if empty then begin
        new(P, rea);
        cellcount(1, 'prod');
        end
    else
        P := freecell;
    with P@ do begin
        ref := 0;
        tag := rea;
        rval := a@.rval * b@.rval;
        end;
    prod := P;
    end
```

```
            rval := a@.rval - b@.rval;
            end;
         subt := S;
         end
      else
         errormsg('subt', 3)  (*Type mismatch*)
   end; (*Function subt*)

(*************************************************************************
 * Function prod                                                         *
 *                                                                       *
 * Purpose: Multiplies two numbers, using the same technique previously  *
 *          described for sum.                                           *
 *                                                                       *
 * Calls: empty, cellcount, freecell, errormsg                           *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function prod(a,b:list):list;
   var  P:list;
   begin
      if (a@.tag =int) and (b@.tag = int) then begin
         if empty then begin
            new(P, int);
            cellcount(1, 'prod');
```

```
if (a@.tag = int) and (b@.tag = int) then begin
   if empty then begin
      new(S, int);
      cellcount(1, 'subt')
      end
   else
      S := freecell;
   with S@ do begin
      ref := 0;
      tag := int;
      ival :=  a@.ival - b@.ival;
      end;
     subt := S;
     end (*if ... int)*)
  else if (a@.tag = rea) and (b@.tag = rea) then begin
     if empty then begin
        new(S, rea);
        cellcount(1, 'subt');
        end
     else
        S := freecell;
     with S@ do begin
        ref := 0;
        tag := rea;
```

```
            R := freecell;
          with R@ do begin
            ref := 0;
            tag := rea;
            R@.rval := a@.rval * b@.rval;
            end;
          sum := R;
          end
        else
          errormsg('sum', 3)  (*Type mismatch*)
    end; (*Function sum*)

(*************************************************************************
 * Function subt                                                         *
 *                                                                       *
 * Purpose: Subtracts two numbers.  Subt alway subtracts the second arg- *
 *          ument from the first.                                        *
 *                                                                       *
 * Calls: empty, cellcount, freecell, errormsg                           *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function subt (a,b:list):list;
  var S:list;
  begin
```

```
 ***************************************************************************)
function sum(a,b: list): list;
   var R,I: list;
   begin
      if (a@.tag = int) and( b@.tag = int) then begin
         if empty then begin
            new(I, int);
            cellcount(1, 'sum');
            end
         else
            I := freecell;
         with I@ do begin
            ref := 0;
            tag :=int;
            I@.ival := a@.ival + b@.ival;
            end;
         sum := I;
         end
      else if (a@.tag = rea) and (b@.tag = rea) then begin
         if empty then begin
            new(R, rea);
            cellcount(1, 'sum');
            end
         else
```

```
            head:= T;
            writeln('In conr 2');
            tail:= nil;
            end;
          P@.tail:= C;
          C@.ref := C@.ref + 1;
          conr := H;
          end;
        (*decr(P);
        decr(C);*)
    end; (*Function conr*)

(*****************************************************************************
 * Function sum                                                              *
 *                                                                           *
 * Purpose: Adds two integers or real numbers, found in the variant          *
 *          fields of the arguments delivered to the function.  The re-      *
 *          sult is returned through a pointer to another cell, created      *
 *          created in the function.                                         *
 *                                                                           *
 * Calls: cellcount, freecell, errormsg                                      *
 *                                                                           *
 * Called by: applyprim                                                      *
```

```
 ***************************************************************************)
function conr(H, T: list): list;
   var P,C: list;
   begin
      if H <> nil then
         H@.ref := H@.ref + 1;
      if T <> nil then
         T@.ref := T@.ref + 1;
      if nullp(H) then
         conr := cons(T, nil)
      else begin
         writeln('In conr 1st call');
         ptrassn(P, H);
         while P@.tail <> nil do
            P:= P@.tail;
         if empty then begin
            new(C,lst);
            cellcount(1, 'conr');
            end
         else
            C := freecell;
         with C@ do begin
            ref := 0;
            tag := lst;
```

```
        C:= freecell;
      with C@ do begin
        ref := 0;
        tag := lst;
        head := nil;
        tail := nil;
        ptrassn(head, H);
        ptrassn(tail, T);
        end;
      cons := C;
    end; (*Function cons*)

(*****************************************************************************
 * Function conr                                                             *
 *                                                                           *
 * Purpose: Same idea as cons except the first argument is made the last     *
 *          element of the second argument (list).  Notice that the          *
 *          technique is different than that used in cons.  Since lists      *
 *          are accessed by pointers to their first elememts, the list       *
 *          tree must be walked down to the last element before the          *
 *          first argument can be added to the end of the list               *
 *                                                                           *
 * Calls: ptrassn, cellcount, ccns, nullp                                    *
 *                                                                           *
 * Called by: applyprim                                                      *
```

```
      decr(L);
      decr(O);
      decr(P);
   end; (*Function initial*)

(*************************************************************************
 * Function cons                                                         *
 *                                                                       *
 * Purpose: Receives two arguments: the first being an atom or list and  *
 *          and the second which must be a list.  Cons takes the first   *
 *          argument and makes it the first element of the second argu-  *
 *          ment (which is a list).                                      *
 *                                                                       *
 * Calls:  ptrassn                                                       *
 *                                                                       *
 * Called by: conr, pairlis, eval, letrec, evlis, applyprim, dcprim      *
 *************************************************************************)
function cons(H,T: list): list;
   var C: list;
   begin
      if empty then begin
         new( C, lst);
         cellcount(1, 'cons');
         end
      else
```

```
 *                                                                    *
 * Purpose: Receives a list as an argument and returns all elememts of *
 *          that list except the last element.                        *
 *                                                                    *
 * Calls: ptrassn, decr, errormsg                                     *
 *                                                                    *
 * Called by: applyprim                                               *
 **********************************************************************)
function initial (L:list): list;
   var O,P: list;
   begin
      if L = nil then
         errormsg('initial', 1)
      else if L@.tag = lst then begin
         ptrassn(P, L);
         repeat
            ptrassn(O, P);
            ptrassn(P, P@.tail);
         until P@.tail = nil;
         O@.tail := nil;
         initial := L
         end
      else
         errormsg('initial', 2);
```

```
  end; (*Function first*)

(*****************************************************************************
 * Function rest                                                             *
 *                                                                           *
 * Purpose: Receives a list as an argument and returns all elements          *
 *          of that list except the first element                            *
 *                                                                           *
 * Calls: errormsg                                                           *
 *                                                                           *
 * Called by: sub, lenp, assic, pairlis, printval, eval, evcon, membp,       *
 *            applyprim, apply                                               *
 *****************************************************************************)
function rest (L: list): list;
  begin
    if L = nil then
        errormsg('rest', 1)
    else if L@.tag = lst then
       rest := L@.tail
    else
       errormsg('rest', 2);
  end; (*Function rest*)

(*****************************************************************************
 * Function initial                                                          *
```

```
            last := L@.head
            end
         else
            errormsg('last', 2);
      end; (*Function last*)

(*************************************************************************
 * Function first                                                        *
 *                                                                       *
 * Purpose: Receives a list as an argument and returns the first element *
 *          of that list.                                                *
 *                                                                       *
 * Calls: None                                                           *
 *                                                                       *
 * Called by: sub, equalp, assoc, pairlis, printval, eval, evcon, membp, *
 *            isfinset, applyprim, apply                                 *
 *************************************************************************)
function first (L:list): list;
   begin
      if L = nil then
         errormsg('first', 1)
      else if L@.tag = lst then
         first := L@.head
      else
         errormsg('first', 2);
```

```
          9: writeln('   Cell for condition is not ''boo''');
         10: writeln('   First element of the list must be an ''alf'' cell');
         11: writeln('   Attribute to be looked up must be in an ''alf''cell');
         12: writeln('   Attempted to take ''repr'' of non finset');
         end; (*case msg of*)
   end; (*Procedure errormsg*)

(*************************************************************************
 * Function last                                                         *
 *                                                                       *
 * Purpose: Receives a list as an argument and returns the last element  *
 *          of that list.                                                *
 *                                                                       *
 * Calls: ptrassn                                                        *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function last (L:list): list;
   begin
      if L = nil then
         errormsg('last', 1)
      else if L@.tag = lst then begin
         repeat
            ptrassn(L, L@.tail);
         until L@.tail = nil;
```

```
 * Function errormsg                                                    *
 *                                                                      *
 * Purpose: Informs the user when errors occur while using the          *
 *          interpreter.  Each error message is annotated to tell where *
 *          it was called from.                                         *
 *                                                                      *
 * Calls: None                                                          *
 *                                                                      *
 * Called by: initial, sum, subt, prod, divi, sub, LT, GT, GE, LE, sinp, *
 *            COSP, TANN, COT, SEC, CSC, ASSOC, EVAL EVCON, ISFINSET,   *
 *            applyprim                                                 *
 ************************************************************************)
procedure errormsg(module: alfa; msg: integer);
  begin
     writeln( 'There is an error in module--> ', module);
     case msg of
        1: writeln('   Cell sent to function is nil');
        2: writeln('   Cell sent to function is not a ''lst'' cell');
        3: writeln('   Types of arguments sent to binary function clash');
        4: writeln('   Index to sub is 0 or not an integer');
        5: writeln('   Cell sent to function must be ''rea'' or ''int''');
        6: writeln('   Value not found in current environment for:');
        7: writeln('   Key word below not recognized by eval');
        8: writeln('   Condition evaluated to nil');
```

```
      else
         errormsg('prod', 3)  (*Type clash*)
   end; (*Function prod*)

(*************************************************************************
 * Function divi                                                         *
 *                                                                       *
 * Purpose: Divides two numbers                                          *
 *                                                                       *
 * Calls: empty, cellcount, freecell, errormsg                           *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function divi(a, b:list):list;
   var D:list;
   begin
      if (a@.tag = int) and (b@.tag = int) then begin
         if empty then begin
            new(D, int);
            cellcount(1, 'divi');
            end
         else
            D := freecell;
         with D@ do begin
            ref := 0;
```

```
            tag := int;
            ival := a@.ival div b@.ival;
          end;
        divi := D;
        end (*if (a@...*)
      else if (a@.tag = rea) and (b@.tag = rea) then begin
        if empty then begin
          new(D, rea);
          cellcount(1, 'divi');
          end
        else
          D := freecell;
        with D@ do begin
          ref := 0;
          tag := rea;
          rval := a@.rval / b@.rval;
          end;
        divi := D;
        end (*else if*)
      else
        errormsg('divi', 3)
    end; (*Function divi*)

(*************************************************************************
 * Function sub                                                          *
```

```
 *                                                                      *
 * Purpose: Receives a list and an integer as arguments.  Returns the   *
 *          element of the list corresponding to the integer, e.g., if  *
 *          the integer is 2, the second element of the list is returned.*
 *                                                                      *
 * Calls: errormsg, rest, first, decr                                   *
 *                                                                      *
 * Called by: applyprim                                                 *
 ************************************************************************)
function sub(L,i:list):list;
   var count: integer;
   begin
      i@.ref := i@.ref + 1;
      if (i@.tag <> int) or (i@.ival = 0) then
         errormsg('sub', 4)
      else begin
         count := i@.ival;
         (*Decrease the size of the list by one until the desired element is
          the first element of a sublist, and return that element*)
         while count <> 1 do begin
            L := rest(L);
            count := count - 1;
            end;
         end;
```

```
      sub := first(L);
      writeln('&&&&&&&&&&&&&&leaving sub');
   end; (*Function sub*)

(*************************************************************************
 * Function  equalp                                                      *
 *                                                                       *
 * Purpose: Works in tandem with the equal function.  Equalp tests for   *
 *          the equality of the arguments sent to it by equalp.  The     *
 *          arguments can be identifiers, numbers, or lists.             *
 *                                                                       *
 * Calls: rest, first, nullp                                             *
 *                                                                       *
 * Called by: equal, membp                                               *
 *************************************************************************)
function equalp(x,y:list):boolean;
   begin
      if (nullp(x)) and (nullp(y)) then
         equalp := true
      else if (x@.tag = alf) and (y@.tag = alf) then
         equalp := x@.aval = y@.aval
      else if (x@.tag = int) and (y@.tag = int) then
         equalp := x@.ival = y@.ival
      else if (x@.tag = rea) and (y@.tag = rea) then
         equalp := x@.rval = y@.rval
```

```
      else if (x@.tag = lst) and (y@.tag = lst) then begin
         if equalp(first(x),first(y)) then
            equalp := equalp(rest(x), rest(y))
         end
      else
         equalp := false
   end; (*Function equalp*)

(*************************************************************************
 * Function  equal                                                       *
 *                                                                       *
 * Purpose: Creates a cell to hold the Boolean response of function      *
 *          equalp.                                                      *
 *                                                                       *
 * Calls: empty, cellcount, freecell, equalp, decr                       *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function equal;
   var E:list;
   begin
      x@.ref := x@.ref + 1;
      y@.ref := y@.ref + 1;
      if empty then begin
         new( E,boo);
```

```
            cellcount(1, 'equal');
            end
         else
            E := freecell;
         with E@ do begin
            ref := 0;
            tag := boo;
            tval := equalp(x,y)
            end;
         equal := E;
         decr(x);
         decr(y);
      end; (*Function equal*)

(*************************************************************************
 * Function LTp (Less Than)                                              *
 *                                                                       *
 * Purpose: Same idea as the equalp function except the first argument   *
 *          argument is tested to see if it is less than the second.     *
 *                                                                       *
 * Calls: errormsg                                                       *
 *                                                                       *
 * Called by:                                                            *
 *************************************************************************)
function LTp(x,y:list):boolean;
```

```
    begin
       if (x@.tag = int) and (y@.tag = int) then
          LTp := x@.ival < y@.ival
       else if (x@.tag = rea) and (y@.tag = rea) then
          LTp := x@.rval < y@.rval
    end; (*Function ITp*)

(*************************************************************************
 * Function LT (Less Than)                                               *
 * Purpose: Creates a cell the hold the Boolean response of calls to     *
 *          LTp.                                                         *
 * Calls: empty, cellcount, freecell, errormsg, decr                     *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function LT(x,y:list):list;
    var A:list;
    begin
       x@.ref := x@.ref + 1;
       y@.ref := y@.ref +1;
       if ((x@.tag = int) and (y@.tag = int))|
          ((x@.tag = rea) and (y@.tag = rea)) then begin
          if empty then begin
             new( A,boo);
             cellcount(1, 'LT');
```

```
            end
         else
            A := freecell;
         with A@ do begin
            ref := 0;
            tag := boo;
            bval := LTp(x,y)
            end;
         LT := A
         end
      else
         errormsg('LT', 3);
      decr(x);
      decr(y);
   end; (*Function LT*)

(*************************************************************************
 * Function GTp (Greater Than)                                           *
 *                                                                       *
 * Purpose: Determines if argument one is greater than argument two.     *
 *                                                                       *
 * Calls: errormsg                                                       *
 *                                                                       *
 * Called by:                                                            *
 *************************************************************************)
```

```
function GTp(x,y:list): boolean;
   begin
      if (x@.tag = int) and (y@.tag = int) then
         GTp := x@.ival > y@.ival
      else if (x@.tag = rea) and (y@.tag = rea) then
         GTp := x@.rval > y@.rval
   end; (*Function GTp*)

(*************************************************************************
 * Function GT (Greater Than)                                            *
 *                                                                       *
 * Purpose: Creates a cell for the Boolean responses of calls to func-   *
 *          tion GTp.                                                    *
 *                                                                       *
 * Calls: empty, cellcount, freecell, errormsg                           *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function GT(x,y:list):list;
   var A:list;
   begin
      if ((x@.tag = int) and (y@.tag = int))|
         ((x@.tag = rea) and (y@.tag = rea)) then begin
         if empty then begin
            new( A, boo);
```

```
            cellcount(1, 'GT');
            end
         else
            A := freecell;
         with A@ do begin
            ref := 0;
            tag := boo;
            bval := GTp(x, y)
            end;
         GT := A
         end (*if*)
      else
         errormsg('GT', 3)
   end; (*Function GT*)

(*************************************************************************
 * Function GEp                                                          *
 *                                                                       *
 * Purpose: Determines if argument 1 is greater than or equal to argu-   *
 *          ment 2.                                                      *
 *                                                                       *
 * Calls: errormsg                                                       *
 *                                                                       *
 * Called by:                                                            *
 *************************************************************************)
```

```
function GEp(x,y:list): boolean;
   begin
      if (x@.tag = int) and (y@.tag = int) then
         GEp := x@.ival >= y@.ival
      else if (x@.tag = rea) and (y@.tag = rea) then
         GEp := x@.rval >= y@.rval
   end; (*Function GEp*)

(*************************************************************************
 * Function GE                                                           *
 *                                                                       *
 * Purpose: Creates a cell to hold Boolean responses from call to func-  *
 *          tion GEp.                                                    *
 *                                                                       *
 * Calls: empty, cellcount, freecell, errormsg                           *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function GE(x,y:list):list;
   var A:list;
   begin
      if ((x@.tag = int) and (y@.tag = int))|
         ((x@.tag = rea) and (y@.tag = rea)) then begin
         if empty then begin
            new(A, boo);
```

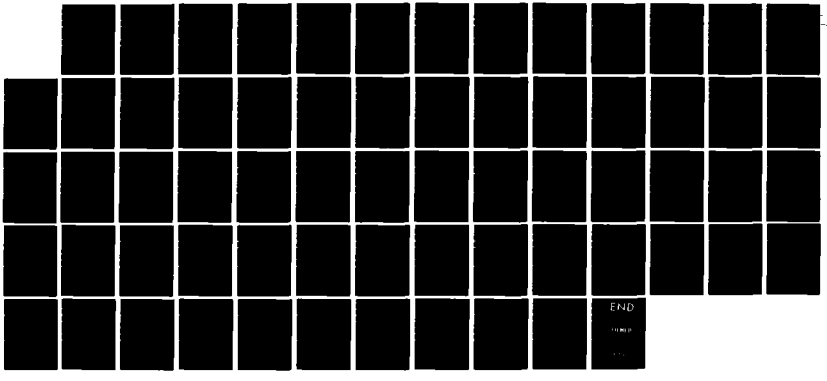
```
            cellcount(1, 'GE');
            end
         else
            A := freecell;
         with A@ do begin
            ref := 0;
            tag := boo;
            bval := GEp(x,y)
            end;
         GE := A
         end
      else
         errormsg('GE', 3)
   end; (*Function GE*)

(*************************************************************************
 * Function LEp                                                          *
 *                                                                       *
 * Purpose: Determines whether argument 1 is less than or equal to argu- *
 *          ment 2                                                       *
 *                                                                       *
 * Calls: errormsg                                                       *
 *                                                                       *
 * Called by:                                                            *
 *************************************************************************)
```

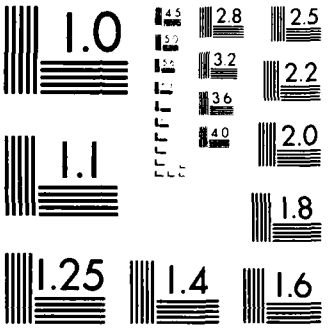
```
function LEp(x,y:list): boolean;
   begin
      if (x@.tag = int) and (y@.tag = int) then
         LEp := x@.ival <= y@.ival
      else if (x@.tag = rea) and (y@.tag = rea) then
         LEp := x@.rval <= y@.rval
   end; (*Function LEp*)

(*************************************************************************
 * Function LE                                                           *
 *                                                                       *
 * Purpose: Creates a cell to hold Boolean responses from calls to func- *
 *          tion LEp.                                                    *
 *                                                                       *
 * Calls: empty, cellcount, freecell, errormsg                           *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function LE(x,y:list):list;
   var A:list;
   begin
      if ((x@.tag = int) and (y@.tag = int)) |
         ((x@.tag = rea) and (y@.tag = rea)) then begin
         if empty then begin
            new(A, boo);
```

```
            cellcount(1, 'LE');
            end
         else
            A := freecell;
         with A@ do begin
            ref := 0;
            tag := boo;
            bval := LEp(x,y)
            end;
         LE := A
         end
      else
         errormsg('IE', 3)
   end; (*Function IE*)

(*************************************************************************
 * Function sinp                                                         *
 *                                                                       *
 * Purpose: Determines the sin of a an angle in degrees.                 *
 *                                                                       *
 * Calls: empty, cellcount, freecell, errormsg                           *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function sinp(x:list):list;
```

AD-A155 218 A PASCAL INTERPRETER FOR THE FUNCTIONAL PROGRAMMING LANGUAGE ELC (EXTENDED LAMBDA CALCULUS)(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA R P STEEN DEC 84 3/3

UNCLASSIFIED F/G 9/2 NL





```
var A:list;
    rad:real;
begin
   if (x@.tag = int) or (x@.tag = rea) then begin
      (*Convert degrees to radians to use built in trig functions*)
      case x@.tag of
         int: rad := (x@.ival * 3.141592654)/180.0;
         rea: rad := (x@.rval * 3.141592654)/180.0;
         end; (*case x@.tag*)
      if empty then begin
         new(A, rea);
         cellcount(1, 'sinp');
         end
      else
         A := freecell;
      with A@ do begin
         ref := 0;
         tag := rea;
         rval := sin(rad);
         end;
      sinp := A;
      end (*if (x@. tag..*)
   else
      errormsg('sinp', 5) (*Can only take the sin of a real or integer*)
```

```
   end; (*Function sinp*)

(*************************************************************************
 * Function cosp                                                         *
 *                                                                       *
 * Purpose: Determines the cosine of an angle given in degrees.          *
 *                                                                       *
 * Calls: empty, cellcount, freecell, errormsg                           *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function cosp(x:list):list;
   var A:list;
       rad:real;
   begin
      if (x@.tag = int) or (x@.tag = rea) then begin
         case x@.tag of
            int: rad := (x@.ival * 3.141592654)/180.0;
            rea: rad := (x@.rval * 3.141592654)/180.0;
            end; (*case x@.tag*)
         if empty then begin
            new(A, rea);
            cellcount(1, 'cosp');
            end
         else
```

```
            A := freecell;
          with A@ do begin
            ref := 0;
            tag := rea;
            rval:= cos(rad);
            end;
          cosp:= A;
          end (*if (x@.tag...*)
        else
          errormsg('cosp', 5) (*Can only take the cos of a real or integer*)
    end; (*Function cosp*)

(*************************************************************************
 * Function tann (Tangent)                                               *
 *                                                                       *
 * Purpose: Determines the tangent of an angle, given in degrees.        *
 *                                                                       *
 * Calls: empty, cellcount, freecell, errormsg                           *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function tann(x:list):list;
    var A:list;
        rad:real;
    begin
```

```
        if (x@.tag = int) or (x@.tag = rea) then begin
           (*Convert degrees to radians*)
           case x@.tag of
              int: rad := (x@.ival * 3.141592654)/180.0;
              rea: rad := (x@.rval * 3.141592654)/180.0;
              end; (*case x@.tag*)
           if empty then begin
              new(A, rea);
              cellcount(1, 'tanr');
              end
           else
              A := freecell;
           with A@ do begin
              ref := 0;
              tag := rea;
              rval:= sin(rad)/cos(rad);
              end;
           tann := A;
           end (*if (x@.tag...*)
        else
           errormsg('tann', 5) (*Can only take the tan of a real or integer*)
     end; (*Function tanr*)

(*************************************************************************
 * Function cot (cotangent)                                              *
```

```
 *                                                                          *
 * Purpose: Determines the cotangent of an angle given in degrees.          *
 *                                                                          *
 * Calls:  empty, cellcount, freecell, errormsg                             *
 *                                                                          *
 * Called by: applyprim                                                     *
 ****************************************************************************)
function cot(x:list):list;
   var A:list;
       rad:real;
   begin
      if (x@.tag = int) or (x@.tag = rea) then begin
         case x@.tag of
            int: rad := (x@.ival * 3.141592654)/180.0;
            rea: rad := (x@.rval * 3.141592654)/180.0;
            end; (*case x@.tag*)
         if empty then begin
            new(A, rea);
            cellcount(1, 'cot');
            end
         else
            A := freecell;
         with A@ do begin
            ref := 0;
```

```
            tag := rea;
            rval:= 1/(sin(rad)/cos(rad));
            end;
         cot:= A;
         end (*if (x@.tag...*)
      else
         errormsg('cot', 5) (*Can only take the cot of a real or integer*)
   end; (*Function cot*)

(*************************************************************************
 * Function sec (Secant)                                                 *
 *                                                                       *
 * Purpose: Determines the secant of an angle given in degrees.          *
 *                                                                       *
 * Calls: empty, cellcount, freecell, errormsg                           *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function sec(x:list):list;
   var A:list;
       rad:real;
   begin
      if (x@.tag = int) or (x@.tag = rea) then begin
         case x@.tag of
            int: rad := (x@.ival * 3.141592654)/180.0;
```

```
            rea: rad := (x@.rval * 3.141592654)/180.0;
            end; (*case x@.tag*)
         if empty then begin
            new(A, rea);
            cellcount(1, 'sec');
            end
         else
            A := freecell;
         with A@ do begin
            ref := 0;
            tag := rea;
            rval := 1/cos(rad);
            end;
         sec := A;
         end (*if (x@.tag...*)
      else
         errormsg('sec', 5) (*Can only take the sec of a real or integer*)
   end; (*Function sec*)

(*************************************************************************
 * Function csc (Cosecant)                                               *
 *                                                                       *
 * Purpose: Determines the cosecant of an angle given in degrees.        *
 *                                                                       *
 * Calls: empty, cellcount, freecell, errormsg                           *
```

```
 *                                                                    *
 * Called by: applyprim                                               *
 **********************************************************************)
function csc(x:list):list;
   var A:list;
       rad:real;
   begin
      if (x@.tag = int) or (x@.tag = rea) then begin
         case x@.tag of
            int: rad := (x@.ival * 3.141592654)/180.0;
            rea: rad := (x@.rval * 3.141592654)/180.0;
            end; (*case x@.tag*)
         if empty then begin
            new(A, rea);
            cellcount(1, 'csc');
            end
         else
            A := freecell;
         with A@ do begin
            ref := 0;
            tag := rea;
            rval := 1/sin(rad);
            end;
         csc := A;
```

```
        end (*if (x@.tag...*)
      else
        errormsg('csc', 5) (*Can only take the csc of a real or integer*)
  end; (*Function csc*)

(*************************************************************************
 * Function atomp                                                        *
 *                                                                       *
 * Purpose: Determines whether the argument sent to it is an atom.       *
 *                                                                       *
 * Calls: None                                                           *
 *                                                                       *
 * Called by: atom, eval                                                 *
 *************************************************************************)
function atomp (L:list): boolean;
  begin
    if L=nil then
       atomp := false
    else
       atomp := L@.tag<>lst;
  end; (*Function atomp*)

(*************************************************************************
 * Function  nullp                                                       *
 *                                                                       *
```

```
 * Purpose: Must receive a list as an argument.  Nullp determines if the *
 *          list is null (empty).                                         *
 *                                                                        *
 * Calls: None                                                            *
 *                                                                        *
 * Called by: conr, null, lenp, assoc, pairlis, evlis, membp              *
 **************************************************************************)
function nullp;
   begin
      nullp := L=nil;
   end; (*Function nullp*)

(**************************************************************************
 * Function  atom                                                         *
 *                                                                        *
 * Purpose: Creates a cell to hold the Boolean responses from calls to    *
 *          function atomp.                                               *
 *                                                                        *
 * Calls: empty, cellcount, freecell, atomp                               *
 *                                                                        *
 * Called by: applyprim                                                   *
 **************************************************************************)
function atom (L:list): list;
   var B: list;
   begin
```

```
      if empty then begin
         new (B, boo);
         cellcount(1, 'atom');
         end
      else B := freecell;
      with B@ do begin
          ref := 0;
          tag := boc;
          bval := atomp(L)
          end;
      atom := B;
   end; (*Function atom*)

(*************************************************************************
 * Function null                                                         *
 *                                                                       *
 * Purpose: Creates a cell to hold the response from a call to function  *
 *          nullp.                                                       *
 *                                                                       *
 * Calls: emptyl cellcount, freecell, nullp                              *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function null (L: list): list;
   var B: list;
```

```
   begin
      if empty then begin
         new( B, boo);
         cellcount(1, 'null');
         end
      else
         B := freecell;
      with B@ do begin
         ref := 0;
         tag := boo;
         bval := nullp(L)
         end;
      null := B;
   end; (*Function null*)

(*****************************************************************************
 * Function  lenp                                                            *
 *                                                                           *
 * Purpose: Determines the length of a list by recursively counting the      *
 *          number of elements.                                              *
 *                                                                           *
 * Calls: nullp, lenp, rest                                                  *
 *                                                                           *
 * Called by: len                                                            *
 *****************************************************************************)
```

```
 *                                                                          *
 * Called by: readval                                                       *
 ************************************ ************************************)
function readint: list;
   var n: integer;
       I: list;
   begin
      n := 0;
      repeat
         n := n*10 + ord(ch) - ord('0');
         if interac then
            read(ch)
         else
            read(infile,ch);
      until (ch<'0') or ('9'<ch) or (ch='.');
      if ch = '.' then
         readint := readrea(n)
      else begin
         if empty then begin
            new( I, int);
            cellcount(1, 'readint');
            end
         else
            I := freecell;
```

```
            read(ch)
         else
            read(infile,ch);
         end; (*while (ch*)
      if empty then begin
         new(R,rea);
         cellcount(1, 'readrea');
         end
      else
         R := freecell;
      with R@ do begin
         ref := 0;
         tag := rea;
         rval := k;
         end;
      readrea := R;
   end; (*Function readrea*)

(*************************************************************************
 * Function readint (Read Integer)                                       *
 *                                                                       *
 * Purpose: Places integers in the proper cells, 'int', as programs are  *
 *          being read, initially.                                       *
 *                                                                       *
 * Calls: readrea, empty, cellcount, freecell                            *
```

```
 *                                                                          *
 * Purpose: Places real numbers in the proper cell as the program is        *
 *          being read.                                                     *
 *                                                                          *
 * Calls: empty, cellcount, freecell                                        *
 *                                                                          *
 * Called by: readint                                                       *
 ***************************************************************************)
function readrea(i:integer): list;
   var R:list;
       k:real;
       n,expo:integer;
   begin
      k:= 0.0;
      expo:=1;
      if interac then
         read(ch)
      else
         read(infile,ch);
      while (ch >= '0') and (ch <= '9') do begin
         n:= n*10 + ord(ch) - ord('0');
         k:= i + (n * (1/(exp(expo * ln(10)))));
         expo := expo +1;
         if interac then
```

```
            read(ch)
         else
            read(infile,ch);
         nonblank;
         if ch = '>' then
            readlist := nil
         else begin
            L:= cons( readval, nil );
            readlist := L;
            nonblank;
            while ch <> '>' do begin
               ptrassn(C, cons( readval, nil));
               ptrassn(L@.tail, C);
               L:= C;
               nonblank;
               end;
            end;
         if interac then
            read(ch)
         else
            read(infile,ch);
      end; (*Function readlist*)

(*************************************************************************
 * Function readrea (Readreal)                                           *
```

```
procedure nonblank;
   begin
      while ch = ' ' do begin
         if interac then
            read(ch)
         else
            read(infile,ch)
         end
   end; (*Procedure nonblank*)

(*************************************************************************
 * Function readlist                                                     *
 *                                                                       *
 * Purpose: At this point a left bracket has been recognized, so the next*
 *          characters are part of a list.  Readlist builds all lists    *
 *          and terminates when a right bracket is recognized.           *
 *                                                                       *
 * Calls: nonblank, ptrassn, readval, decr                               *
 *                                                                       *
 * Called by: readval                                                    *
 *************************************************************************)
function readlist: list;
   var L, C: list;
   begin
      if interac then
```

```
                i := 1;
                while not(ls@.aval(.i.) = ' ') do begin
                   write(ls@.aval(.i.));
                   i := i+1;
                   end;
                end;
             boo: begin
                if diags then
                   write(' /tag is boo, ref = ',ls@.ref:1,' /');
                write(ls@.bval);
                end
          end;
    end; (*Procedure printval*)

function readval: list; forward;

(*************************************************************************
 * Function nonblank                                                     *
 *                                                                       *
 * Purpose: Passes by blanks when reading input either from a terminal   *
 *          or a file until the next character is read.                  *
 * Calls:                                                                *
 *                                                                       *
 * Called by: readlist                                                   *
 *************************************************************************)
```

```
            printval(first(L));
            L:= rest(L);
            if L <> nil then begin
               write(', ');
               if diags then
                  writeln;
               end
         until L = nil;
         write('>');
         writeln;
         end;
      int: begin
         if diags then
             write(' /tag is int,ref = ',ls@.ref:1,' /');
         write(ls@.ival:6);
         end;
      rea: begin
         if diags then
            write(' /tag is rea,ref = ',ls@.ref:1,' /');
         write(ls@.rval:6:6);
         end;
      alf: begin
         if diags then
            write(' /tag is alf, ref= ',ls@.ref:1,' /');
```

```
 *          tail of the 'lst' cells until the atom cells are found.      *
 *          When an atom cell is found, the contents of it are immediate-*
 *          ly printed out.                                              *
 *                                                                       *
 * Calls: first, rest                                                    *
 *                                                                       *
 * Called by: main program                                               *
 *************************************************************************)
procedure printval;
   var L: list;
       i: integer;
   begin
      if ls = nil then
         write('<>')
      else
         case ls@.tag of
            lst: begin
               L:= ls;
               write('<');
               if diags then begin
                  write(' /tag is lst,ref= ',ls@.ref:1,' /');
                  writeln;
                  end;
               repeat
```

```
            writeln;
            writeln('IN function pairlis we are linking');
            writeln('  v is below');
            printval(v);
            writeln;
            writeln('  DO you want to try to read x? y/n');
            readln(ans);
            if ans = 'y' then
               printval(x);
            end; (*if diags*)
         if nullp(v) then
            pairlis := A
         else
            pairlis:= cons( cons( first(v), cons( first(x), nil)),
                            pairlis( rest(v), rest(x), A));
         decr(v);
         decr(x);
         decr(A);
      end; (*Function pairlis*)

(*************************************************************************
 * Procedure printval                                                    *
 *                                                                       *
 * Purpose: Prints the contents of any list by starting at the top of the*
 *          list tree and recursively calling itself on the head and tail*
```

```
      decr(t);
   end; (*Function assoc*)

(*************************************************************************
 * Function pairlis                                                      *
 *                                                                       *
 * Purpose: Given a list of bound variable, v, and a list of actual      *
 *          values, x, pairlis binds the variables with the actuals by   *
 *          creating lists of pairs (attribute value pairs). These       *
 *          lists of pairs are then added to the current environment, A, *
 *          for use in the evaluation of a function call.                *
 *                                                                       *
 * Calls: nullp, cons, first, decr, rest                                 *
 *                                                                       *
 * Called by: eval, apply                                                *
 *************************************************************************)
function pairlis (v, x, A: list) :list;
   begin
      if v <> nil then
         v@.ref := v@.ref + 1;
      if x <> nil then
         x@.ref := x@.ref + 1;
      if A <> nil then
         A@.ref := A@.ref + 1;
      if diags then begin
```

```
            if diags then begin
               (*Print out trace information*)
               writeln;
               writeln('IN function assoc, looking for the value for ');
               printval(x);
               writeln;
               writeln('DO you want to try to see what was found? y/n');
               writeln('if x is a recursive function;it can''t be printed');
               readln(ans);
               if ans = 'y' then
                  printval( first(rest(first(r))));
               writeln;
               end; (*if diags*)
            assoc := first(rest(first(r)));
            end (*if found*)
         else begin
             errormsg('assoc', 6);
             printval(x);
             writeln;
             end
      end;
   decr(A);
   decr(x);
   decr(r);
```

```
    r, t: list;
    found: boolean;
begin
   if A <> nil then
      A@.ref := A@.ref + 1;
   if x <> nil then
      x@.ref := x@.ref + 1;
   if nullp(x) then
      errormsg('assoc', 1)
   else if x@.tag<>alf then
      errormsg('assoc', 11)
   else begin
      v:= x@.aval;
      ptrassn(r, A);
      found := false;
      while not nullp(r) and not found do begin
         ptrassn(t, first(first(r)) );
         u := t@.aval;
         if u=v then found := true
            else begin
               ptrassn(r, rest(r) );
               end
         end; (*while not*)
       if found then begin
```

```
            cellcount(1, 'len');
            end
         else
            A := freecell;
         with A@ do begin
            ref := 0;
            tag := int;
            ival := lenp(L)
            end;
         len := A;
      end; (*Function len*)

(*************************************************************************
 * Function  assoc                                                       *
 *                                                                       *
 * Purpose: Given an association list, 'A', and an attribute, x, find    *
 *          the value for x, e.g., for A = <x, 'value'>, 'value' would   *
 *          be returned.                                                 *
 *                                                                       *
 * Calls: nullp, errormsg, ptrassn, first, rest, decr                    *
 *                                                                       *
 * Called by:                                                            *
 *************************************************************************)
function assoc (A,x: list): list;
   var u, v: alfa;
```

```
function lenp(L:list):integer;
   var length:integer;
   begin
      length := 0;
      if nullp(L) then
         length := 0
      else
         length := 1 + (lenp(rest(L)));
      lenp := length
   end; (*Function lenp*)

(*************************************************************************
 * Function  len                                                         *
 *                                                                       *
 * Purpose: Creates a 'int' cell to hold the result of a call to lenp.   *
 *                                                                       *
 * Calls: empty, cellcount, freecell, lenp                               *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function len(L:list):list;
   var A:list;
   begin
      if empty then begin
         new(A, int);
```

```
        with I@ do begin
            ref := 0;
            tag := int;
            ival := n
            end;
        readint := I;
        end(*else begin*)
    end; (*Function readint*)

(*************************************************************************
 * Function readstring                                                   *
 *                                                                       *
 * Purpose: Reads quoted strings and places them in the proper cells     *
 *          ('alf'), as the program is read.                             *
 *                                                                       *
 * Calls:  cellcount, freecell                                           *
 *                                                                       *
 * Called by: readval                                                    *
 *************************************************************************)
function readstring: list;
    var a: alfa;
        i: integer;
        A: list;
    begin
       if interac then
```

```
    read(ch)
 else
    read(infile,ch);
 i := 1;
 a:= '  ';
 while ch <> '''' do begin
    a(.i.) := ch;
    i := i+1;
    if interac then
       read(ch)
    else
       read(infile,ch)
    end;
 if empty then begin
    new( A, alf);
    cellcount(1, 'readstr');
    end
 else
    A := freecell;
 with A@ do begin
    ref := 0;
    tag := alf;
    aval := a
    end;
```

```
      if interac then
         read(ch)
      else
         read(infile,ch);
      readstring := A;
   end; (*Function readstring*)

(*************************************************************************
 * Function digit                                                        *
 *                                                                       *
 * Purpose: Boolean function to recognize digits.                        *
 *                                                                       *
 * Calls: None                                                           *
 *                                                                       *
 * Called by: readident                                                  *
 *************************************************************************)
function digit(ch: char):boolean;
   type digits = set of char;
   var digitset : digits;
   begin
      digitset := (.'0'..'9'.);
      digit := ch in digitset;
   end; (*Function digit*)

(*************************************************************************
```

```
 * Function letter                                                          *
 *                                                                          *
 * Purpose: Boolean function to recognize upper and lower case letters      *
 *                                                                          *
 * Calls:                                                                   *
 *                                                                          *
 * Called by: readident                                                     *
 ****************************************************************************)
function letter(ch: char):boolean;
   type letters = set of char;
   var letterset : letters;
   begin
      letterset]:= (.'a'..'z','A'..'Z'.);
      letter := ch in letterset;
   end; (*Function letter*)

(****************************************************************************
 * Function readident                                                       *
 *                                                                          *
 * Purpose: Places identifiers (up to 10 letters, characters or combina-    *
 *          tion), in the proper cell as a program is being read.           *
 *                                                                          *
 * Calls: cellcount, freecell, empty                                        *
 *                                                                          *
 * Called by: readval                                                       *
```

```
 ***************************************************************************)
function readident: list;
   var a: alfa;
       i: integer;
       A: list;
   begin
      i := 1;
      a := '  ';
      while letter(ch)|digit(ch) do begin
         a(.i.) := ch;
         i:=i+1;
         if interac then
            read(ch)
         else
            read(infile,ch);
      end; (*while ch*)
      if empty then begin
         new(A, alf);
         cellcount(1, 'readiden');
         end
      else
         A := freecell;
      with A@ do begin
         ref := 0;
```

```
            tag := alf;
            aval := a;
            end;
         readident := A;
      end; (*Function readident*)

(*************************************************************************
 * Function readval                                                      *
 *                                                                       *
 * Purpose: Readval is the first function called, as a program is read.  *
 *          Readval recognized the first letter of the program and deter-*
 *          mines if a list, integer, real, or identifier is being read. *
 *                                                                       *
 * Calls: readlist, readstring, readident, readint                       *
 *                                                                       *
 * Called by: readlist, main program                                     *
 *************************************************************************)
function readval;
   begin
      if ch = '<' then
         readval := readlist
      else if ch ='''' then
         readval := readstring
      else if letter(ch) then
         readval := readident
```

```
      else
         readval := readint
   end; (*Function readval*)

function evcon (L,a: list): list; forward;
function evlis (L,a: list): list; forward;
function apply (f,x: list): list; forward;
function letrec(f, lam, bdy, a: list):list; forward;

(*************************************************************************
 * Function eval                                                         *
 *                                                                       *
 * Purpose: Main decoding function of the interpreter.  Strips the first *
 *          element from the expression and determines how the expression*
 *          is to be evaluated.                                          *
 *                                                                       *
 * Calls: atomp, ptrassn, first, rest, assoc, letrec, cons, empty        *
 *        cellcount, freecell, apply, evlis, eval, evcon, pairlis, decr, *
 *        errormsg                                                       *
 *                                                                       *
 * Called by: eval, letrec, evlis                                        *
 *************************************************************************)
function eval (e, a: list): list;
   var T,C,e1p: list;
       e1:alfa;
```

```
begin
if diags then begin
   (*Print out trace information*)
   writeln;
   writeln('IN function eval, the expression being evaluated is: ');
   printval(e);
   writeln;
   end;
(*writeln('Entering eval the ref cnt to e is ',e@.ref);
writeln('              ''''                    a is ',a@.ref);*)
if e <> nil then
   e@.ref := e@.ref + 1;
if a <> nil then
   a@.ref := a@.ref + 1;
if atomp(e) then eval := e
else begin
   ptrassn(e1p, first(e));
   e1 := e1p@.aval;
   if e1 = 'list' then eval := evlis(rest(e), a)
   else if e1 = 'finset' then eval := e
   else if e1 = 'con' then eval := first( rest(e))
   else if e1 = 'var' then eval := assoc( a, first( rest(e)))
   else if e1 = 'letrec' then eval := letrec(first( rest(e)),
      first(rest(rest(e))), first(rest(rest(rest(e)))), a)
```

```
else if e1 = 'lambda' then begin
   if empty then begin
      new(C, alf);
      cellcount(1, 'eval');
      end
   else
      C := freecell;
   with C@ do begin
      ref := 0;
      tag := alf;
      aval := 'closure';
      end;
   eval := cons( cons(C,e), cons(a,nil) );
   end (*if e1 = 'lambda'*)
else if e1 = 'if' then eval := evcon( rest(e), a)
else if e1 = 'call' then
 eval :=apply(eval( first(rest(e)), a), evlis( rest(rest(e)), a))
else if e1 = 'apply' then
 eval:=apply(eval(first(rest(e)),a),eval(first(rest(rest(e))),a))
else if e1 = 'let' then begin
   (*if diags then begin
      writeln('rest(e)');
      printval(rest(e));
      writeln;
```

```
        writeln('first(rest(e))');
        printval(first(rest(e)));
        writeln;
        writeln('first(rest(first(rest(e))))');
        printval(first(rest(first(rest(e)))));
        writeln;
        writeln('Other part of pairlis below');
        printval(first(first(rest(e))));
        writeln;
        writeln('B is below');
        printval(first(rest(rest(first(rest(e))))));
        writeln;
        end;*)
  (*First, evaluate actual parameters and then form the environment
     of evaluation for the let statement*)
    T:= pairlis(evlis(first( first( rest(e))), a),
                evlis( first(rest( first( rest(e)))), a), a) ;
    writeln('The reference cnt of T is  ',T@.ref);
    eval := eval( first( rest (rest (first(rest(e))))), T);
    end
else begin
    errormsg('eval', 7);
    writeln(e1);
    writeln;
```

```
            end
         end;
         decr(e);
         decr(a);
         decr(e1p);
      end (*Function eval*);

(*************************************************** ******************
 * Function letrec                                                     *
 *                                                                     *
 * Purpose: Builds an environment for a recursive function.            *
 *                                                                     *
 * Calls: empty, cellcount, freecell, ptrassn, cons, decr              *
 *                                                                     *
 * Called by: eval                                                     *
 ***********************************************************************)
function letrec;
   var S,C,B,Z,L,M:list;
   begin
      writeln('Entering letrec ref cnts for f,lam,bdy,a ',f@.ref);
      writeln('                                         ',lam@.ref);
      writeln('                                         ',bdy@.ref);
      writeln('                                         ',a@.ref);
      if f <> nil then
         f@.ref := f@.ref + 1;
```

```
if lam <> nil then
   lam@.ref := lam@.ref + 1;
if bdy <> nil then
   bdy@.ref := bdy@.ref + 1;
if a <> nil then
   a@.ref := a@.ref + 1;
(*This procedure creates the proper environment for the evaluation
of a recursive function call and evaluates the body of the letrec.
The environment must include the body of the function itself
tagged with the name of the recursive function.*)
if diags then begin
   writeln;
   writeln('IN function letrec');
   (*writeln('   f is below ');
   printval(f);*)
   writeln;
   writeln('   lam is below');
   (*printval(lam);
   writeln;
   writeln('   bdy is below');
   printval(bdy);*)
   writeln;
   end; (*if diags*)
if empty then begin
```

```
   new(L, lst); cellcount(1, 'letrec') end
else L := freecell;
if empty then begin
   new(B, lst); cellcount(1, 'letrec') end
else B := freecell;
if empty then begin
   new(Z, lst); cellcount(1, 'letrec') end
else Z := freecell;
if empty then begin
   new(C, alf); cellcount(1, 'letrec') end
else C := freecell;
if empty then begin
   new(M, lst); cellcount(1, 'letrec') end
else M := freecell;
if empty then begin
   new(S, lst); cellcount(1, 'letrec') end
else S := freecell;
(*Create an attribute value pair with the function name as the attribute
 and a closure as the value and add this to the current environment.
 The unique thing about this closure is that the ep part of the closure
 points back to the front of the current environment to enable a recurs-
 ive call of the function*)
with C@ do begin
   ref := 0;
```

```
        tag:= alf;
        aval:= 'closure';
        end;
    with L@ do begin
        ref := 1;
        tag:= lst;
        ptrassn(head, Z);
        ptrassn(tail, a);
        end;
    with M@ do begin
        ref := 0;
        tag:= lst;
        ptrassn(head, L);
        tail:= nil;
        end;
    with B@ do be in
        ref := 0;
        tag:= lst;
        ptrassn(head, cons(C, lam));
        ptrassn(tail, M);
        end;
    with S@ do begin
        ref := 0;
        tag := lst;
```

```
   end; (*Function apply*)

(*****************************************************************************
 * Function dcprim (Declare Primitive)                                       *
 *                                                                           *
 * Purpose: Builds an environment for primitive function calls.              *
 *                                                                           *
 * Calls: cellcount, cons                                                    *
 *                                                                           *
 * Called by:                                                                *
 *****************************************************************************)
function dcprim (primname:alfa; primitives:list): list;
   var C,D,E: list;
   begin
      (*At this point, there is no reason to check the free list since this
       is the very start of the program*)
      new (C, alf);
      new (D, alf);
      new (E, alf);
      cellcount(3, 'dcprim');
      with C@ do begin
         ref := 0;
         tag := alf;
         aval := primname;
         end;
```

```
      writeln('IN function apply; the function is: ');
      printval(first(f));
      writeln;
      writeln('   in apply the arguments are:');
      printval(x);
      writeln;
      end; (*if diags*)
   ptrassn(f1p, first(f));
   f1 := f1p@.aval;
   if f1 = 'prim' then
      apply:=applyprim(rest(f), x)
   else begin
      temp := eval( first(rest(rest(rest(first(f))))),
         pairlis( first(rest(rest(first(f)))), x, first(rest(f)) ));
      if callonly or diags then begin
         writeln('   in apply results below:');
         printval(temp);
         writeln;
         end;
      apply := temp;
      end;
   decr(f);
   decr(x);
   (*decr(f1p);*)
```

```
 * Function apply (Apply any function to its arguments)                    *
 *                                                                          *
 * Purpose: Determines whether the body of a 'call' expression is a         *
 *          primitive call or an abstraction (closure).  If the function    *
 *          is a primitive, the function name and the arguments are sent    *
 *          to function applyprim for evaluation. If a closure is de-       *
 *          tected, actual values are bound to variables and added to       *
 *          the current envircnment before evaluation.                      *
 *                                                                          *
 * Calls: ptrassn, applyprim, first, rest, pairlis, decr                    *
 *                                                                          *
 * Called by: eval                                                          *
 ***************************************************************************)
function apply;
   var f1p, temp: list;
       f1: alfa;
   begin
      writeln('IN apply f coming in its ref cnt is ',f@.ref);
      if f <> nil then
         f@.ref := f@.ref + 1;
      if x <> nil then
         x@.ref := x@.ref + 1;
      if callonly or diags then begin
         writeln;
```

```
      else if f1 = 'GT' then
         applyprim := GT(first(x), first(rest(x)))
      else if f1 = 'LT' then
         applyprim := LT(first(x), first(rest(x)))
      else if f1 = 'GE' then
         applyprim := GE(first(x), first(rest(x)))
      else if f1 = 'LE' then
         applyprim := LE(first(x), first(rest(x)))
      else if f1 = 'sin' then
         applyprim := sinp(first(x))
      else if f1 = 'cos' then
         applyprim := cosp(first(x))
      else if f1 = 'tan' then
         applyprim := tann(first(x))
      else if f1 = 'cot' then
         applyprim := cot(first(x))
      else if f1 = 'sec' then
         applyprim := sec(first(x))
      else if f1 = 'csc' then
         applyprim := csc(first(x));
      decr(f);
      decr(x);
   end; (*Function applyprim*)

(*************************************************************************
```

```
   applyprim := cons(first(x), first(rest(x)))
   end
else if f1 = 'conr' then
   applyprim := conr(first(x), first(rest(x)))
else if f1 = 'atom' then
   applyprim := atom( first(x))
else if f1 = 'null' then
   applyprim := null( first(x))
else if f1 = 'len' then
   applyprim := len( first(x))
else if f1 = 'sum' then
   applyprim := sum(first(x), first(rest(x)))
else if f1 = 'subt' then
   applyprim := subt(first(x), first(rest(x)))
else if f1 = 'prod' then
   applyprim := prod(first(x), first(rest(x)))
else if f1 = 'divi' then
   applyprim := divi(first(x), first(rest(x)))
else if f1 = 'sub' then
   applyprim := sub(first(x), first(rest(x)))
else if f1 = 'memb' then
   applyprim := memb(first(x), first(rest(x)))
else if f1 = 'equal' then
   applyprim := equal(first(x), first(rest(x)))
```

```
      writeln('IN applyprim this is going to rest');
      printval(first(x));
      writeln;
      end;
   applyprim := rest( first(x))
   end
else if f1 = 'repr' then begin
   if isfinset(x) then
      applyprim:= rest( first(x))
   else
      errormsg('apprim', 12)
   end
else if f1 = 'initial' then
   applyprim := initial( first(x))
else if f1 = 'cons' then begin
   if diags then begin
      writeln('Sending the next 2 lines to cons');
      printval (first(x));
      writeln;
      writeln;
      printval(first(rest(x)));
      writeln;
      writeln;
      end;
```

```
if f <> nil then
   f@.ref := f@.ref + 1;
if x <> nil then
   x@.ref := x@.ref + 1;
if callonly or diags then begin
   writeln;
   writeln('IN function applyprim; the function delivered is: ');
   printval(f);
   writeln;
   writeln('   in applyprim x is:');
   printval(x);
   writeln;
   writeln;
   end;
ptrassn(f1p, first(f));
f1:=f1p@.aval;
if f1 = 'id' then
   applyprim := first(x)
else if f1 = 'first' then
   applyprim := first( first(x))
else if f1 = 'last' then
   applyprim := last( first(x))
else if f1 = 'rest' then begin
   if diags then begin
```

```
            (*x is a finite set if the first element is 'finset'*)
            isfinset := x1@.aval = 'finset'
        end
    end; (*Function isfinset*)

(*************************************************************************
 * Function applyprim (Apply primitive function)                         *
 *                                                                       *
 * Purpose: Determines the primitive function to be used (f) and applies *
 *          it to the arguments (x), e.g., as in standard function no-   *
 *          tation f(x).                                                 *
 *                                                                       *
 * Calls: ptrassn, first, last, rest, repr, errormsg, initial, cons, sub,*
 *        memb, equal, subt, prod, divi, GT, LT, LE, GE, sin, cos, tan,  *
 *        cot, sec, csc, decr, conr, atom, null, len, sum                *
 *                                                                       *
 * Called by: apply                                                      *
 *************************************************************************)
function applyprim (f, x:list): list;
    var  f1p:list;
         f1:alfa;
    begin
       writeln;
       writeln('IN applyprim f refcnt is ',f@.ref);
       writeln('IN applyprim x refcnt is ',x@.ref);
```

```
            end;
         memb := C;
         end
   end; (*Function memb*)

(*************************************************************************
 * Function isfinset                                                     *
 *                                                                       *
 * Purpose: Determines whether a list is a finite set by checking the    *
 *          first element of the list.                                   *
 *                                                                       *
 * Calls: first, errormsg,                                               *
 *                                                                       *
 * Called by: applyprim                                                  *
 *************************************************************************)
function isfinset (x:list): boolean;
   var x1: list;
   begin
      if x = nil then
         errormsg('isfinset', 1)
      else begin
         x1 := first(first(x));
         if x1@.tag <> alf then
            errormsg('isfinset', 10)
         else
```

```
 * Function memb (Member)                                                  *
 *                                                                         *
 * Purpose: Creates a cell to hold the Boolean response from a call to     *
 *          function membp.                                                *
 * Calls: empty, cellcount, freecell, membp                                *
 *                                                                         *
 * Called by: applyprim                                                    *
 ***************************************************************************)
function memb(x, L:list): list;
   var C:list;
   begin
      if L = nil then
         errormsg('memb', 1)
      else begin
         if empty then begin
            new(C, boo);
            cellcount(1,'memb');
            end
         else
            C := freecell;
         with C@ do begin
            ref := 0;
            tag := boo;
            bval := membp(x, L);
```

```
      else
         evlis := cons( eval( first(L), a), evlis( rest(L), a));
   end; (*Function evlis*)

(*************************************************************************
 * Function membp                                                        *
 *                                                                       *
 * Purpose: Works in tandem with function memb to determine if x is a    *
 *          member of list L.                                            *
 *                                                                       *
 * Calls: nullp, equalp, first, rest                                     *
 *                                                                       *
 * Called by: memb                                                       *
 *************************************************************************)
function membp(x, L:list):boolean;
   begin
      if nullp(L) then
         membp := false
      else if equalp(x, first(L)) then
         membp := true
      else
         membp:= membp(x, rest(L))
   end; (*Function membp*)

(*************************************************************************
```

```
        else begin
            (*Evaluate false consequent*)
            if diags then begin
               writeln;
               writeln('in evcon evaluating false consequent next');
               end;
            evcon := eval( first(rest(rest(first(L)))), a);
            end;
         decr(B);
      end; (*Function evcon*)

(*************************************************************************
 * Function evlis (Evaluate List)                                        *
 *                                                                       *
 * Purpose: Evaluates a list of arguments and constructs a list of the   *
 *          of the results.                                              *
 *                                                                       *
 * Calls: nullp, cons, eval, evlis                                       *
 *                                                                       *
 * Called by: eval, evlis                                                *
 *************************************************************************)
function evlis;
   begin
      if nullp(L) then
          evlis := nil
```

```
 * Called by: eval                                                          *
 ***************************************************************************)
function evcon;
   var B: list;
   begin
      (*Evaluate condition*)
      B:= eval( first(first(L)), a);
      if diags then begin
         writeln;
         writeln('IN function evcon the condition evaluated to: ');
         printval(B);
         end;
      if B=nil then
          errormsg('evcon', 8)
      else if B@.tag <> boo then
         errormsg('evcon', 9)
      else if B@.bval then begin
         (*Evaluate true consequent*)
         if diags then begin
            writeln;
            writeln('in evcon evaluating true consequent next');
            end;
         evcon := eval( first(rest(first(L))), a);
         end
```

```
            ptrassn(head, B);
            tail := nil;
            end;
         with Z@ do begin
            ref := 0;
            tag := lst;
            ptrassn(head, f);
            ptrassn(tail, S);
            end;
         (*Evaluate the call to the recursive function in the created
          environment*)
         letrec := eval(bdy, L);
      end; (*Function letrec*)

(*************************************************************************
 * Function evcon (Evaluate Conditional)                                 *
 *                                                                       *
 * Purpose: Evaulates a conditional expression (consists of three        *
 *          sublists: conditicn, true conseguent, and false conse-       *
 *          guent) by first determining the result of the condition      *
 *          and then evaluating the true conseguent or the false conse-  *
 *          guent.                                                       *
 *                                                                       *
 * Calls: eval, errormsg, first, rest, decr                              *
 *                                                                       *
```

```
      with D@ do begin
         ref := 0;
         tag := alf;
         aval := 'prim';
         end;
      with E@ do begin
         ref := 0;
         tag := alf;
         aval := primname;
         end;
      dcprim := cons(cons(C,cons(cons(D,cons(E,nil)),nil)),primitives);
   end; (*Function dcprim*)

(*************************************************************************
 * Function readfname (Read file name)                                   *
 *                                                                       *
 * Purpose: Reads the name of a file that contains an ELC program.  The  *
 *          filename is input interactively and may be up to eighty      *
 *          eighty characters long.                                      *
 *                                                                       *
 * Calls: None                                                           *
 *                                                                       *
 * Called by: main program                                               *
 *************************************************************************)
function readfname: filename;
```

```
    var c: char;
        temp: filename;
        i: integer;
    begin
       i := 1;
       while c <> ' ' do begin
          read(c);
          temp (.i.) := c;
          i := i + 1;
          end;
       readfname := temp
    end; (*readfname*)

begin (*Main Program*)
    newcells := 0;
    (*Initialize the header record of the freelist*)
    with hdr do begin
       numcells := 0;
       next := nil;
       end;
    (*Initialize the array for the cell creation information*)
    for k := 1 to 26 do begin
          with counts(.k.) do begin
          modul := 'empty';
          cellcnt := 0;
```

```
      end;
   end;
(*Build the association list for the primitive operations*)
primitives := nil;
primitives := dcprim('first', primitives);
primitives := dcprim('last', primitives);
primitives := dcprim('initial', primitives);
primitives := dcprim('rest', primitives);
primitives := dcprim('cons', primitives);
primitives := dcprim('conr', primitives);
primitives := dcprim('atom', primitives);
primitives := dcprim('null', primitives);
primitives := dcprim('sum', primitives);
primitives := dcprim('subt', primitives);
primitives := dcprim('prod', primitives);
primitives := dcprim('divi', primitives);
primitives := dcprim('sul', primitives);
primitives := dcprim('equal', primitives);
primitives := dcprim('len', primitives);
primitives := dcprim('GT', primitives);
primitive:     dcprim('LT', primitives);
primitives := dcprim('GE', primitives);
primitives := dcprim('LE', primitives);
primitives := dcprim('sin', primitives);
```

```
primitives := dcprim('cos', primitives);
primitives := dcprim('tan', primitives);
primitives := dcprim('cot', primitives);
primitives := dcprim('sec', primitives);
primitives := dcprim('csc', primitives);
primitives := dcprim('id', primitives);
primitives := dcprim('repr', primitives);
primitives := dcprim('memb', primitives);
primitives@.ref := 1;
writeln;
diags := false;
interac := false;
ch := ' ';
(*Ask user initial questions to see if a trace is requested.*)
date(dtmoyr);
time(curtime);
writeln('Functional Language Interpreter (ELC): ');
writeln('  Version 12.0');
writeln('  date/time ',dtmcyr,',',curtime);
writeln;
(*Determine whether the program is to be interactively typed in or read
 from a file by checking the number of arguments on the command line.
 If there are two arguments and the second one is 'i' the session is to
 be interactive, otherwise, prompt the user for the filename (1..80 char *)
```

```
if argc = 1 then begin
   writeln('File ELC program to be read from: ');
   filearg := readfname;
   reset(infile, filearg);
   end
else if argc = 2 then begin
   argv(1, filearg);
   if filearg = 'i ' then begin
      interac := true;
      writeln('Note: ''!'' ends an interactive terminal session');
      end
   else
      reset(infile, filearg)
   end;
writeln('Total program trace ? type in ''t''');
writeln('Monitor function calls only? type in ''c''');
writeln('  If no trace, type any other key');
readln(ans);
if ans = 't' then diags := true
else if ans = 'c' then callonly := true;
while ch <> '!' do
   begin
      writeln;
      writeln('Enter Expression');
```

```
        nonblank;
        if ch <> '!' then begin
           temp := readval;
           if diags then begin
              writeln('Expression after reading');
              printval(temp);
              writeln;
              end;
           printval( eval( temp,primitives));
           end;
        writeln(' ');
     end;
writeln('Evaluation Completed');
writeln;
(*Print the amount of user and system time taken by the program*)
writeln('*************************************************');
writeln('                              Statistics');
(*Sysclock and clock are built-in Berkeley Pascal function to show
   System time = The amount of time that was spent inside the Unix
                 system itself, doing work on behalf of the user's
                 program.
   User time = The amount of time actually spent obeying the instruc-
               tions in the user's program*)
writeln('System time was ',sysclock:10,' milliseconds');
```

```
    writeln('User time was   ',clock:10,' milliseconds');
    l:= 1;
    writeln;
    writeln('+------------------------------+');
    writeln('|Module          Cells created  |');
    writeln('|------          -------------  |');
    (*Print out the contents of the array containing the cell creation
     information.*)
    while counts (.l.).modul <> 'empty' do begin
       with counts(.l.) do
          writeln('|',modul,'     ',cellcnt,'      |');
       l := l+1;
       end;
    writeln('+------------------------------+');
    writeln(' Total cells    ',newcells);
    writeln;
end. (*Program func*)
```

LIST OF REFERENCES

1. Naval Postgraduate School Technical Report NPS52-81-012, Elements of Programming Linguistics, Part I, The Lambda Calculus and its Implementation, by B. J. MacLennan, Monterey, CA, August 1981.

2. Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," Communications of the ACM, V21, no. 8, 1978.

3. Stone, H. S., Introduction to Computer Architecture, Science Research Associates, 1980.

4. Darlington, J., Functional Programming and Its Applications. An Advanced Course, Cambridge University Press, England, 1982.

5. Stanford University Technical Report STAN-CS-73-403, Hints on Programming Language Design, by C. A. R. Hoare, Palo Alto, CA, December 1973.

6. MacLennan, B. J., Functional Programming Methodology: Theory and Practice, To be published by Addison Wesley.

7. Early, G., "Functional Programming and the Two-Pass Assembler," Sigplan Notices, V17, No. 8, 1982.

8. Douglas, John, H., "New Computer Architectures Tackle Bottleneck," High Technology, June 1983.

9. Cooper, D., Oh! Pascal!, W. W. Norton and Company, 1982.

10. MacLennan, B. J., Principles of Programming Languages, Holt, Rhinehart and Winston, 1983.

BIBLIOGRAPHY

Baden, S., "Berkely FP Experiences with a Functional Programming Language," Spring Compcon 83, Intellectual Leverage for the Information Society, IEEE, 1983.

Boecker, H., "Functional Programming in Basic-Plus," Computers in Education, North-Holland Publishing Company, 1981.

Berkling, K., "A Consistent Extension of the Lambda-Calculus as a Base for Functional Programming Languages," Information and Control, V55, 1982.

Durham, T., "Asserting Assertional Style," Computing, March 1983.

Fehr, Elfriede, "The Simplest Functional Programming Language is Neither Simple nor Functional," SIGPLAN Notices, V18, 4, April, 1983.

Gram, C., "Easy Functional Programming," Norddata 81, V1, 1981.

Inoue, K., "Implementation of a Functional Programming Language FPL," Transcripts of the Institute for Electronics and Communications Engineering Japan, VE65, May 1982.

Islam, N., "Transforming Functional Programs," Micro-Delcon 82: The Delaware Bay Computer Conference, IEEE, 1982.

Kennaway, J. R., Parallel Implementation of Functional Languages, Proceedings of the 1982 International Conference on Parallel Processing, 1982.

MacLennan, B. J., "Values and Objects in Programming Languages," SIGPLAN Notices, V17, 12, December, 1982.

Nagata, M., "An Approach to Construction of Functional Programs," Journal Of Information Processing, V5, No. 4, 1982.

Organick, E. I., "New Directions in Computer Systems Architecture," Euromicro, Journal 5, 1979.

Spector, David, "The Simplest Functional Programming Language," SIGPLAN Notices, V18, 1, January 1983.

Stabile, L. A., FP and Its Use as a Command Language, Proceedings of Distributed Computing, Compcon 80, Twenty-First IEEE Computer Society International Conference, 1980.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Captain Ralph P. Steen Jr., USA 465 West Waterloo Street Canal Winchester, Ohio 43110	2
4.	Curricular Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943	1

END

FILMED

7-85

DTIC