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
A Pascal Interpreter for the Functional Programming Language

Ralph P. Steen, Jr.

Naval Postgraduate School
Monterey, California 93943

Naval Postgraduate School
Monterey, California 93943

December 1984

255

UNCLASSIFIED

Approved for public release; distribution is unlimited

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

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.
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.
Approved by: B.J. MacLennan, Thesis Advisor
Gordon Bradley, Second Reader
B.J. MacLennan, Chairman, Department of Computer Science
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 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 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

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.</td>
<td>PURPOSE AND BACKGROUND</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>A. PURPOSE</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>B. BACKGROUND</td>
<td>8</td>
</tr>
<tr>
<td>II.</td>
<td>INTERPRETER OPERATIONS</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>A. ASSUMPTIONS</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>B. PASCAL IS THE IMPLEMENTATION LANGUAGE</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>1. Pascal Does Have Some Advantages</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>2. Pascal's Disadvantages are Commonly Known</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>C. THE CELL AND THE REPRESENTATION OF ATOMS AND LISTS</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>1. Atoms</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>2. What about lists?</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>D. ELC PROGRAMS</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>1. Definition</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>2. Reading Programs</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>3. Primitive Operations</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>E. THE HEART OF THE INTERPRETER, THE EVAL FUNCTION</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>1. Function Eval's Keywords</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>2. Printing Results</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>F. MEMORY MANAGEMENT</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>1. Overview</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>2. Conventions</td>
<td>38</td>
</tr>
<tr>
<td>III.</td>
<td>HUMAN INTERFACE WITH THE INTERPRETER</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>A. LOADING A PROGRAM</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>B. EXECUTION TRACE</td>
<td>41</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1.1 Factorial Program in FP and PL/I ............. 12
1.2 Sum in Three Functional Languages .............. 13
2.1 ELC Lists .................................. 17
2.2 Representation of a Cell Containing an Integer .. 18
2.3 Pascal Representation of a List Cell ............. 19
2.4 Representation of a Simple List ................. 19
2.5 List Within a List ........................... 20
2.6 Function Eval ................................ 26
2.7 Primitives Association List ...................... 27
2.8 Lambda Expression ............................ 27
2.9 Using Call to Invoke Functions ................... 28
2.10 ELC Conditional .............................. 31
2.11 Letrec Environment ............................ 33
2.12 Doubling Function ............................. 34
2.13 Function Sum ................................ 36
2.14 Function Ptransn .............................. 37
I. PURPOSE AND BACKGROUND

A. PURPOSE

The purpose of this thesis is 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 oper-
ations. Conventional programming 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 Calculator 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 conven-
tional 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 assign-
ment statements to achieve the desired results, instead of
cconcerning himself with the problem as a whole. Another
problem with the assignment statement is that it makes
programs unreliable. Mathematical proofs do not lend them-
selves 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 possess mathematical properties. Backus [Ref. 2] has even shown that the functional language FP lends itself to an algebra of programs that can allow for relatively 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, whereas the FP assembler occupied 249 lines, of which more than 100 lines were only a single char. so as to aid in program readability and clarity. This fact would 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
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

\[ \text{<list a b c>} \]

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.
  \[ \text{repr} \langle \text{finset} : b \ c \ 1 \ 2 \rangle = \langle a \ b \ c \ 1 \ 2 \rangle \]

- **Len** Determines the length of any list.
  \[
  \begin{align*}
  \text{len} \langle a \rangle & = 1 \\
  \text{len} \langle \rangle & = 0 \\
  \text{len} \langle a \ b \ c \ d \rangle & = 3
  \end{align*}
  \]

- **Equal** Equal tests the equality of any two atoms or any two lists.
  \[
  \begin{align*}
  \text{equal} \langle 2 \ 2 \rangle & = \text{true} \\
  \text{equal} \langle a \ c \rangle & = \text{false} \\
  \text{equal} \langle a \ b \ c \rangle & = \text{true} \\
  \text{equal} \langle a \ b \ c \rangle & = \text{false}
  \end{align*}
  \]

- **GT** Greater-than tests if arg1 is greater than arg2.
  \[
  \begin{align*}
  \text{GT} \ arg1 \ arg2 & = \text{true/false} \\
  \text{GT} \ 2 \ 3 & = \text{false} \\
  \text{GT} \ 5 \ 1 & = \text{true}
  \end{align*}
  \]

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.
  \[
  \begin{align*}
  \text{memb} \ arg1 \ \langle \text{arg2} \rangle & = \text{true/false} \\
  \text{memb} \ 2 \ \langle 1 \ 2 \ 3 \rangle & = \text{true}
  \end{align*}
  \]

24
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 of 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 list 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, e.g., 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.

  \[
  \text{cons} \; a \; <b \; c> = <a \; b \; c>
  \]

  \[
  \text{cons} \; <a \; b> \; <c \; d> = <<a \; b> \; c \; d>
  \]

- **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 1st cells that are at the top of the Figure, there are three with the tail field of the 1st 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 1st cell, however, does not point to a leaf or information cell, it points to another 1st 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. EIC PROGRAMS

1. Definition

An EIC 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.

\[
\text{printval \ (eval( readval, primitives))}
\]

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 '"', transferring 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
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.
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 '1st' 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 list 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 representation 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 are 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 to Cooper [Ref. 9] for 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 programming? 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].
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
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 negligible. 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, e1: list;
  e1: alpha;
  begin
    if atom (e) then eval := e
    else begin
      e1p := first (e);
      e1p := eval (a);
      if e1 = 'list' then eval := evalis (rest (e), a)
      else if e1 = 'inset' 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, alpha);
        cellcount (1, 'eval');
        with C do begin
          tag := alpha;
          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),
        evalis (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 (evalis (first (first (rest (e)))), a),
        evalis (first (first (rest (rest (e)))), a), a)
        eval := eval (first (rest (first (rest (rest (e))))), T);
      end
      else errormsg ('eval');
    end (function eval);
  end

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:
Figure 2.7 Primitives Association List.

\[ \text{<con 1> = 1} \]
\[ \text{<con <1 2 3>> = <1 2 3>} \]

- **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.

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'.

\[
\text{Call to Primitive Function}
\]
\[
\text{Call to User Defined Function}
\]

**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.

\[
\text{Call to Function}
\]
- '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 evalis which in turn sends each of the elements of the list to function eval with the current environment. Evalis 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.

\[
\text{<Call <lambda <x> <call <var sum> <var x> <var x>><con 5>}}
\]

- 'call' recognized

- \text{<lambda <x> <call <var sum> <var x> <var x>><con 5>}} sent to eval with current environment.

- 'lambda' recognized and a closure is formed as follows \text{<closure lambda <x> <call <var sum> <var x> <var x>><con 5>}} a where a is the current environment.

- \text{<con 5> is sent to 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 \text{<call <var sum> <var x> <var x>><con 5> is sent to function eval. But an environment must be created before evaluation can be completed.}

- \text{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 \text{<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 invoke all functions.

\[
\text{if } \langle\text{call } \langle\text{var equal} \rangle \langle\text{con 0} \rangle \langle\text{con 3} \rangle \rangle \\
\langle\text{con 'true'} \rangle \\
\langle\text{con 'false'} \rangle
\]

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,

\[
\langle\text{call } \langle\text{var equal} \rangle \langle\text{con 0} \rangle \langle\text{con 3} \rangle \rangle
\]

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 occur. 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 of the last sublist. In the example, since 0 and 3 are not equal the condition 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 for 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 needed 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

  \[
  \text{<call} \text{<lambda<x> <call <var sum> \text{<var x> <var x>>> <con 5>}}
  \]

  the let statement allows you to write the expression in Figure 2.12. In general, \text{<let} \text{<<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
accomplished using a single lambda expression as in Figure 2.8, but the let statement makes the function call expression

\[
\text{<call \ <var \ double> \ <con 2>>}
\]

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

\[ \text{apply \ var \ sum} \ \text{list \ con} \ 2 \ \text{con} \ 3 \\\n\]

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 '1st', 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 '1st' 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 so 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 are 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 later. 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(I, '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(R, 'sum');
    end
    else
      R := freecell;
      with R do begin
        ref := 0;
        tag := rea;
        R rval := x.ival + y.ival;
      end;
    sum := R;
  end
  else
    errorsy ('sum') [Type mismatch]
end; {Function sum}

'@' Indicates Pointer

Figure 2.13 Function Sum.

Pointers with other pointers by using an assignment state-
ment 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:

- \texttt{ftrassn} Overwrites pointers
- \texttt{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.

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`.

Figure 2.14  Function `ptrassn`.

Cells `x` and `y` before entering `ptrassn`
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 new cells 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 reading since it is never sent to the printout 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 generated 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 'cons' 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 cons 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 cons>
      <call <var reverse>
      <call <var rest> <var L>>>
      <call <var sub> <var L> <con 1>>>>>>>
```
Statistics
System time was 366 milliseconds
User time was 6600 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldcprim</td>
<td>94</td>
</tr>
<tr>
<td>cons</td>
<td>402</td>
</tr>
<tr>
<td>readiden</td>
<td>57</td>
</tr>
<tr>
<td>readint</td>
<td>1</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>null</td>
<td>11</td>
</tr>
</tbody>
</table>

Total cells 561

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>21</td>
<td>printval</td>
</tr>
<tr>
<td>36</td>
<td>505</td>
<td>cellcount</td>
</tr>
<tr>
<td>63</td>
<td>1477</td>
<td>nullp</td>
</tr>
<tr>
<td>84</td>
<td>3123</td>
<td>first</td>
</tr>
<tr>
<td>94</td>
<td>1692</td>
<td>rest</td>
</tr>
<tr>
<td>122</td>
<td>402</td>
<td>cons</td>
</tr>
<tr>
<td>258</td>
<td>10</td>
<td>sub</td>
</tr>
<tr>
<td>539</td>
<td>190</td>
<td>atomp</td>
</tr>
<tr>
<td>547</td>
<td>1477</td>
<td>nullp</td>
</tr>
<tr>
<td>564</td>
<td>11</td>
<td>null</td>
</tr>
<tr>
<td>595</td>
<td>24</td>
<td>assoc</td>
</tr>
<tr>
<td>643</td>
<td>33</td>
<td>pairlis</td>
</tr>
<tr>
<td>663</td>
<td>21</td>
<td>printval</td>
</tr>
</tbody>
</table>
<call <var append>
  <list <con g><con h> <con j>>
  <list <con g> <con r><con s>>>

Results of Append Function (Compiled Interpreter)

Enter Expression

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

Evaluation Completed

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

Statistics
System time was 183 milliseconds
User time was 616 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>discrim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>402</td>
</tr>
<tr>
<td>readiden</td>
<td>57</td>
</tr>
<tr>
<td>readint</td>
<td>1</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>null</td>
<td>11</td>
</tr>
</tbody>
</table>

Total cells 561

Results of Append (ELC Interpreter interpreted by Berkeley Pascal)

Enter Expression

<var a, l, 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
  <lamda <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.>
      <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|mem|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> ee1 <list> ee1
<list> ee1 <lookup var>
<const. exp.> ee1 <const. exp.>
<letter> ::= a..z|A..Z
<number> ::= <digit>ee1|<digit>ee1 '.' <dijit>ee1
<digit> ::= <0..9>
<atom> ::= letter|number|identifier

48
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>'

47
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.

\[
\text{<call <var cons>}
\text{<list a b>}
\text{<list c d e>}
\]

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

\[
\text{<if <<call <boolean exp.>}
\text{<True consequent>}
\text{<False consequent>>>}
\]

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 getting 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 3 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 surprising because there is one less layer of software involved 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 management 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 3 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 surprising 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 execution 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 compiled version runs much faster as seen by the time of execution statistics located in Appendix 3, 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 another 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>!</call>

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

-----------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>323</td>
</tr>
<tr>
<td>readiden</td>
<td>42</td>
</tr>
<tr>
<td>readint</td>
<td>1</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>null</td>
<td>11</td>
</tr>
<tr>
<td>consr</td>
<td>9</td>
</tr>
</tbody>
</table>
-----------------------------------
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
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>jdcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>323</td>
</tr>
<tr>
<td>readiden</td>
<td>42</td>
</tr>
<tr>
<td>readint</td>
<td>1</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>null</td>
<td>11</td>
</tr>
<tr>
<td>conr</td>
<td>9</td>
</tr>
</tbody>
</table>

Total cells 476

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>func</td>
</tr>
<tr>
<td>33</td>
<td>11</td>
<td>printval</td>
</tr>
<tr>
<td>36</td>
<td>420</td>
<td>cellcount</td>
</tr>
<tr>
<td>63</td>
<td>1349</td>
<td>nullp</td>
</tr>
<tr>
<td>84</td>
<td>2861</td>
<td>first</td>
</tr>
<tr>
<td>94</td>
<td>1545</td>
<td>rest</td>
</tr>
<tr>
<td>122</td>
<td>323</td>
<td>cons</td>
</tr>
<tr>
<td>137</td>
<td>10</td>
<td>conr</td>
</tr>
<tr>
<td>258</td>
<td>10</td>
<td>sub</td>
</tr>
<tr>
<td>539</td>
<td>169</td>
<td>atomp</td>
</tr>
<tr>
<td>547</td>
<td>1349</td>
<td>nullp</td>
</tr>
<tr>
<td>564</td>
<td>11</td>
<td>null</td>
</tr>
<tr>
<td>599</td>
<td>33</td>
<td>assoc</td>
</tr>
<tr>
<td>643</td>
<td>22</td>
<td>pairlis</td>
</tr>
<tr>
<td>663</td>
<td>11</td>
<td>printval</td>
</tr>
<tr>
<td>702</td>
<td>67</td>
<td>readval</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>704</td>
<td>91</td>
<td>nonblank</td>
</tr>
<tr>
<td>714</td>
<td>24</td>
<td>readlist</td>
</tr>
<tr>
<td>770</td>
<td>1</td>
<td>readint</td>
</tr>
<tr>
<td>827</td>
<td>167</td>
<td>digit</td>
</tr>
<tr>
<td>835</td>
<td>210</td>
<td>letter</td>
</tr>
<tr>
<td>843</td>
<td>42</td>
<td>readident</td>
</tr>
<tr>
<td>867</td>
<td>67</td>
<td>readval</td>
</tr>
<tr>
<td>879</td>
<td>11</td>
<td>evcon</td>
</tr>
<tr>
<td>880</td>
<td>135</td>
<td>evlis</td>
</tr>
<tr>
<td>881</td>
<td>52</td>
<td>apply</td>
</tr>
<tr>
<td>882</td>
<td>1</td>
<td>letrec</td>
</tr>
<tr>
<td>884</td>
<td>169</td>
<td>eval</td>
</tr>
<tr>
<td>947</td>
<td>1</td>
<td>letrec</td>
</tr>
<tr>
<td>1012</td>
<td>11</td>
<td>evcon</td>
</tr>
<tr>
<td>1041</td>
<td>135</td>
<td>evlis</td>
</tr>
<tr>
<td>1093</td>
<td>41</td>
<td>applyprim</td>
</tr>
<tr>
<td>1198</td>
<td>52</td>
<td>apply</td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
<td>dcprim</td>
</tr>
<tr>
<td>1261</td>
<td>1</td>
<td>readfname</td>
</tr>
</tbody>
</table>
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

-----------------------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>---------------</td>
</tr>
<tr>
<td>dcprom</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>383</td>
</tr>
<tr>
<td>readiden</td>
<td>47</td>
</tr>
<tr>
<td>readint</td>
<td>1</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
</tbody>
</table>

57
null

------------------------
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 94 |
icons 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.js
Profiled Fri Dec 14 22:46 1984

Line Count
1 1 func
33 11 printval
36 47e 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   | dcprom         |
| 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

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

60
System time was 216 milliseconds
User time was 683 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>445</td>
</tr>
<tr>
<td>readidem</td>
<td>52</td>
</tr>
<tr>
<td>readint</td>
<td>10</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>eval</td>
<td>11</td>
</tr>
<tr>
<td>null</td>
<td>11</td>
</tr>
<tr>
<td>sing</td>
<td>10</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>445</td>
</tr>
<tr>
<td>readidem</td>
<td>52</td>
</tr>
<tr>
<td>readint</td>
<td>10</td>
</tr>
</tbody>
</table>

61
| letrec | 6 | |
| eval   | 11 | |
| null   | 11 | |
| sinp   | 10 | |

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total cells</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td>629</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>func</td>
</tr>
<tr>
<td>33</td>
<td>11</td>
<td>printval</td>
</tr>
<tr>
<td>36</td>
<td>573</td>
<td>cellcount</td>
</tr>
<tr>
<td>63</td>
<td>1666</td>
<td>nulp</td>
</tr>
<tr>
<td>84</td>
<td>3579</td>
<td>first</td>
</tr>
<tr>
<td>94</td>
<td>1959</td>
<td>rest</td>
</tr>
<tr>
<td>122</td>
<td>445</td>
<td>cons</td>
</tr>
<tr>
<td>411</td>
<td>10</td>
<td>sinp</td>
</tr>
<tr>
<td>539</td>
<td>212</td>
<td>atomp</td>
</tr>
<tr>
<td>547</td>
<td>1666</td>
<td>nulp</td>
</tr>
<tr>
<td>564</td>
<td>11</td>
<td>null</td>
</tr>
<tr>
<td>593</td>
<td>104</td>
<td>assoc</td>
</tr>
<tr>
<td>643</td>
<td>44</td>
<td>pairlis</td>
</tr>
<tr>
<td>663</td>
<td>11</td>
<td>printval</td>
</tr>
<tr>
<td>702</td>
<td>103</td>
<td>readval</td>
</tr>
<tr>
<td>704</td>
<td>144</td>
<td>nonblank</td>
</tr>
<tr>
<td>714</td>
<td>41</td>
<td>readlist</td>
</tr>
<tr>
<td>773</td>
<td>10</td>
<td>readint</td>
</tr>
<tr>
<td>827</td>
<td>217</td>
<td>digit</td>
</tr>
<tr>
<td>835</td>
<td>279</td>
<td>letter</td>
</tr>
<tr>
<td>843</td>
<td>52</td>
<td>readident</td>
</tr>
</tbody>
</table>

62
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>367</td>
<td>103</td>
<td>readval</td>
<td></td>
</tr>
<tr>
<td>879</td>
<td>11</td>
<td>evcon</td>
<td></td>
</tr>
<tr>
<td>880</td>
<td>167</td>
<td>evlis</td>
<td></td>
</tr>
<tr>
<td>881</td>
<td>73</td>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>882</td>
<td>1</td>
<td>letrec</td>
<td></td>
</tr>
<tr>
<td>884</td>
<td>212</td>
<td>eval</td>
<td></td>
</tr>
<tr>
<td>947</td>
<td>1</td>
<td>letrec</td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td>11</td>
<td>evcon</td>
<td></td>
</tr>
<tr>
<td>1041</td>
<td>167</td>
<td>evlis</td>
<td></td>
</tr>
<tr>
<td>1093</td>
<td>51</td>
<td>applyprim</td>
<td></td>
</tr>
<tr>
<td>1198</td>
<td>73</td>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
<td>dcprim</td>
<td></td>
</tr>
<tr>
<td>1261</td>
<td>1</td>
<td>readfname</td>
<td></td>
</tr>
</tbody>
</table>
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 <L> <<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 13>]
    >>>>>!

Results of halving function (compiled)
```
Enter Expression
< 2, 3, 4, 5, 10, 11, 12, 13, 14, 15>

Evaluation Completed

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

-------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>674</td>
</tr>
<tr>
<td>readiden</td>
<td>37</td>
</tr>
<tr>
<td>readint</td>
<td>11</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>eval</td>
<td>15</td>
</tr>
<tr>
<td>null</td>
<td>11</td>
</tr>
<tr>
<td>divi</td>
<td>10</td>
</tr>
</tbody>
</table>
-------------------
Total cells     898

Results of Evaluating (interpreted)
Enter Expression
< 2, 3, 4, 5, 10, 11, 12, 13, 14, 15>

Evaluation Completed

******************************************************************************
Statistics
System time was 483 milliseconds
User time was 10183 milliseconds

---

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcpri</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>674</td>
</tr>
<tr>
<td>readidn</td>
<td>87</td>
</tr>
<tr>
<td>readint</td>
<td>11</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>eval</td>
<td>15</td>
</tr>
<tr>
<td>null</td>
<td>11</td>
</tr>
<tr>
<td>livi</td>
<td>10</td>
</tr>
</tbody>
</table>

---

Total cells 898

Profile for halving function (use of bu).

Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Red Dec 12 12:48 1984 test11.6
Profiled Thu Dec 13 10:03 1984

Line    Count   func
1       1       printval
33      11      celcount
36      842     nullp
63      2057    first
84      4420    rest
94      2494    cons
122     674     livi
233     10      atomp
539     305     nullp
547     2057    null
564     11      assoc
595     166     }
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>543</td>
<td>103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>663</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>702</td>
<td>168</td>
<td></td>
<td></td>
</tr>
<tr>
<td>704</td>
<td>233</td>
<td></td>
<td></td>
</tr>
<tr>
<td>714</td>
<td>70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>770</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>827</td>
<td>350</td>
<td></td>
<td></td>
</tr>
<tr>
<td>835</td>
<td>448</td>
<td></td>
<td></td>
</tr>
<tr>
<td>843</td>
<td>87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>867</td>
<td>168</td>
<td></td>
<td></td>
</tr>
<tr>
<td>875</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>880</td>
<td>240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>881</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>882</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>884</td>
<td>305</td>
<td></td>
<td></td>
</tr>
<tr>
<td>947</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1041</td>
<td>240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1093</td>
<td>51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1198</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1261</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

pair lis
print val
read val
non blank
read list
read int
digit
letter
read ident
read val
ev con
evlis
apply
letrec
eval
letrec
ev con
evlis
apply
dcprim
read fname

67
| equal  | 1806 |   |
| sum    | 255  |   |
| len    | 411  |   |

Total cells 57682

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>func</td>
</tr>
<tr>
<td>33</td>
<td>19</td>
<td>printval</td>
</tr>
<tr>
<td>36</td>
<td>57626</td>
<td>cellcount</td>
</tr>
<tr>
<td>62</td>
<td>1806</td>
<td>equal</td>
</tr>
<tr>
<td>63</td>
<td>664983</td>
<td>nullp</td>
</tr>
<tr>
<td>84</td>
<td>1404922</td>
<td>first</td>
</tr>
<tr>
<td>94</td>
<td>743026</td>
<td>rest</td>
</tr>
<tr>
<td>122</td>
<td>43559</td>
<td>cons</td>
</tr>
<tr>
<td>153</td>
<td>255</td>
<td>sum</td>
</tr>
<tr>
<td>293</td>
<td>2267</td>
<td>equalp</td>
</tr>
<tr>
<td>291</td>
<td>1806</td>
<td>equal</td>
</tr>
<tr>
<td>539</td>
<td>70799</td>
<td>atomp</td>
</tr>
<tr>
<td>547</td>
<td>664983</td>
<td>nullp</td>
</tr>
<tr>
<td>564</td>
<td>4554</td>
<td>null</td>
</tr>
<tr>
<td>576</td>
<td>1233</td>
<td>lenp</td>
</tr>
<tr>
<td>587</td>
<td>411</td>
<td>len</td>
</tr>
<tr>
<td>593</td>
<td>36403</td>
<td>assoc</td>
</tr>
<tr>
<td>643</td>
<td>10367</td>
<td>pairlis</td>
</tr>
<tr>
<td>663</td>
<td>10</td>
<td>printval</td>
</tr>
<tr>
<td>702</td>
<td>530</td>
<td>readval</td>
</tr>
<tr>
<td>704</td>
<td>744</td>
<td>nonlink</td>
</tr>
<tr>
<td>714</td>
<td>214</td>
<td>readlist</td>
</tr>
</tbody>
</table>

81
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 | |
|-----------|---------------|
| 9         | 84            |
| 16         | 49559         |
| 36         | 313           |
| 39         | 3             |
| 42         | 80            |
| 46         | 1             |
| 45         | 4554          |
| 46         | 666           |
| 47         | 1             |

40
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

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>decprim</td>
<td>34</td>
</tr>
<tr>
<td>icons</td>
<td>49559</td>
</tr>
<tr>
<td>readlen</td>
<td>313</td>
</tr>
<tr>
<td>readint</td>
<td>3</td>
</tr>
<tr>
<td>nulrec</td>
<td>30</td>
</tr>
<tr>
<td>eval</td>
<td>1</td>
</tr>
</tbody>
</table>

75
<if <call <var equal>
 <var k>
 <call <var first>
  <call <var first> <var T>>>
 <call <var first>
  <call <var rest> <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>
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 valuable to see how many times a person's name appears in a newspaper to gain some insight into how important they might be.

(letrec lam
  (lambda (L)
    (if (null L)
        (con null)
        (cons (first L)
              (letrec isfinfunc
                (lambda (T)
                  (if (null T)
                      (con true)
                      (if (equal (len T)
                                    (first T))
                          (con 2)
                          (if (equal (len T)
                                        (first T))
                              (call (var len)
                                    (call (var first) T))
                              (call (var mem)
                                    (call (var first)
                                          (call (var lam)
                                                (call (var null) L)))))))))
  L)))
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>94</td>
<td>1148</td>
<td>rest</td>
</tr>
<tr>
<td>122</td>
<td>395</td>
<td>cons</td>
</tr>
<tr>
<td>137</td>
<td>3</td>
<td>conr</td>
</tr>
<tr>
<td>258</td>
<td>3</td>
<td>sub</td>
</tr>
<tr>
<td>411</td>
<td>3</td>
<td>sinp</td>
</tr>
<tr>
<td>539</td>
<td>123</td>
<td>atomp</td>
</tr>
<tr>
<td>547</td>
<td>988</td>
<td>nullp</td>
</tr>
<tr>
<td>564</td>
<td>8</td>
<td>null</td>
</tr>
<tr>
<td>599</td>
<td>61</td>
<td>assoc</td>
</tr>
<tr>
<td>643</td>
<td>23</td>
<td>pairlis</td>
</tr>
<tr>
<td>663</td>
<td>4</td>
<td>printval</td>
</tr>
<tr>
<td>702</td>
<td>154</td>
<td>readval</td>
</tr>
<tr>
<td>704</td>
<td>219</td>
<td>nonblank</td>
</tr>
<tr>
<td>714</td>
<td>65</td>
<td>readlist</td>
</tr>
<tr>
<td>770</td>
<td>4</td>
<td>readint</td>
</tr>
<tr>
<td>827</td>
<td>377</td>
<td>digit</td>
</tr>
<tr>
<td>835</td>
<td>466</td>
<td>letter</td>
</tr>
<tr>
<td>843</td>
<td>85</td>
<td>readident</td>
</tr>
<tr>
<td>867</td>
<td>154</td>
<td>readval</td>
</tr>
<tr>
<td>873</td>
<td>8</td>
<td>evcon</td>
</tr>
<tr>
<td>880</td>
<td>96</td>
<td>evlis</td>
</tr>
<tr>
<td>881</td>
<td>39</td>
<td>apply</td>
</tr>
<tr>
<td>882</td>
<td>2</td>
<td>letrec</td>
</tr>
<tr>
<td>884</td>
<td>123</td>
<td>eval</td>
</tr>
<tr>
<td>947</td>
<td>2</td>
<td>letrec</td>
</tr>
<tr>
<td>1012</td>
<td>8</td>
<td>evcon</td>
</tr>
<tr>
<td>1041</td>
<td>96</td>
<td>evlis</td>
</tr>
<tr>
<td>1093</td>
<td>29</td>
<td>applyprim</td>
</tr>
<tr>
<td>1198</td>
<td>39</td>
<td>apply</td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
<td>dcpriam</td>
</tr>
<tr>
<td>1261</td>
<td>1</td>
<td>readfname</td>
</tr>
</tbody>
</table>
Results of composition (interpreted)
Enter Expression
<1.000000, 0.866025, 0.707107>
Evaluation Completed

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

----------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dc_prim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>395</td>
</tr>
<tr>
<td>readilen</td>
<td>85</td>
</tr>
<tr>
<td>readint</td>
<td>4</td>
</tr>
<tr>
<td>letrec</td>
<td>12</td>
</tr>
<tr>
<td>eval</td>
<td>2</td>
</tr>
<tr>
<td>null</td>
<td>8</td>
</tr>
<tr>
<td>conr</td>
<td>2</td>
</tr>
<tr>
<td>sfp</td>
<td>3</td>
</tr>
</tbody>
</table>
----------------------------------
Total cells 595

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>4</td>
</tr>
<tr>
<td>56</td>
<td>539</td>
</tr>
<tr>
<td>63</td>
<td>588</td>
</tr>
<tr>
<td>84</td>
<td>2111</td>
</tr>
</tbody>
</table>
\texttt{\textless \text{call \textless \text{var rest} \textgreater} \\
\textless \text{var l} \textgreater \textgreater \textgreater} \\
\texttt{\textless \text{let \textless \text{dot} \textless \text{lambda \textless f1 f2} \textgreater} \\
\textless \text{lambda \textless x} \textgreater \\
\texttt{\textless \text{call \textless \text{var f1} \textless \text{call \textless \text{var f2} \textless \text{var x} \textgreater} \textgreater} \textgreater \textgreater} \\
\texttt{\textless \text{call} \\
\texttt{\textless \text{call \textless \text{var dot} \textless \text{var mapsin} \textless \text{var reverse} \textgreater} \\
\textless \text{list \textless \text{con 45} \textless \text{con 60} \textless \text{con 90} \textgreater \textgreater} \textgreater} !}

Results of composition (compiled)

Enter Expression
\texttt{\langle 1.000000, 0.866025, 0.707107 \rangle}

Evaluation Completed

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

\textbf{Statistics}
\texttt{System time was 250 milliseconds}
\texttt{User time was 616 milliseconds}

\begin{tabular}{|l|}
\hline
Module & Cells created \\
\hline
\texttt{\textbar dcpri m} & 84 \\
\texttt{\textbar cons} & 335 \\
\texttt{\textbar readlen} & 85 \\
\texttt{\textbar readint} & 4 \\
\texttt{\textbar letrec} & 12 \\
\texttt{\textbar eval} & 2 \\
\texttt{\textbar null} & 8 \\
\texttt{\textbar cons} & 2 \\
\texttt{\textbar sin} & 3 \\
\hline
\end{tabular}

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 cons>
            <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 >

72
<table>
<thead>
<tr>
<th>Line</th>
<th>Column</th>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>663</td>
<td>12</td>
<td>printval</td>
<td></td>
</tr>
<tr>
<td>702</td>
<td>150</td>
<td>readval</td>
<td></td>
</tr>
<tr>
<td>704</td>
<td>205</td>
<td>nonblank</td>
<td></td>
</tr>
<tr>
<td>714</td>
<td>55</td>
<td>readlist</td>
<td></td>
</tr>
<tr>
<td>770</td>
<td>15</td>
<td>readint</td>
<td></td>
</tr>
<tr>
<td>827</td>
<td>332</td>
<td>digit</td>
<td></td>
</tr>
<tr>
<td>835</td>
<td>427</td>
<td>letter</td>
<td></td>
</tr>
<tr>
<td>843</td>
<td>80</td>
<td>readident</td>
<td></td>
</tr>
<tr>
<td>867</td>
<td>150</td>
<td>readval</td>
<td></td>
</tr>
<tr>
<td>879</td>
<td>32</td>
<td>evcon</td>
<td></td>
</tr>
<tr>
<td>880</td>
<td>260</td>
<td>evlis</td>
<td></td>
</tr>
<tr>
<td>881</td>
<td>93</td>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>882</td>
<td>1</td>
<td>letrec</td>
<td></td>
</tr>
<tr>
<td>884</td>
<td>335</td>
<td>eval</td>
<td></td>
</tr>
<tr>
<td>947</td>
<td>1</td>
<td>letrec</td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td>32</td>
<td>evcon</td>
<td></td>
</tr>
<tr>
<td>1041</td>
<td>260</td>
<td>evlis</td>
<td></td>
</tr>
<tr>
<td>1093</td>
<td>82</td>
<td>applyprim</td>
<td></td>
</tr>
<tr>
<td>1198</td>
<td>93</td>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
<td>dcprim</td>
<td></td>
</tr>
<tr>
<td>1261</td>
<td>1</td>
<td>readfname</td>
<td></td>
</tr>
<tr>
<td>Module</td>
<td>Cells created</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jdcFrim</td>
<td>84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>icons</td>
<td>531</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ireadiden</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>readint</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>null</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LE</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total cells 748

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>12</td>
<td>printval</td>
</tr>
<tr>
<td>36</td>
<td>692</td>
<td>cellcount</td>
</tr>
<tr>
<td>63</td>
<td>2566</td>
<td>nullp</td>
</tr>
<tr>
<td>84</td>
<td>5526</td>
<td>first</td>
</tr>
<tr>
<td>94</td>
<td>2921</td>
<td>rest</td>
</tr>
<tr>
<td>122</td>
<td>531</td>
<td>cons</td>
</tr>
<tr>
<td>258</td>
<td>30</td>
<td>sub</td>
</tr>
<tr>
<td>384</td>
<td>10</td>
<td>LEp</td>
</tr>
<tr>
<td>394</td>
<td>10</td>
<td>LE</td>
</tr>
<tr>
<td>535</td>
<td>335</td>
<td>atomp</td>
</tr>
<tr>
<td>547</td>
<td>2566</td>
<td>nullp</td>
</tr>
<tr>
<td>564</td>
<td>22</td>
<td>null</td>
</tr>
<tr>
<td>599</td>
<td>166</td>
<td>assoc</td>
</tr>
<tr>
<td>643</td>
<td>33</td>
<td>pairlis</td>
</tr>
</tbody>
</table>

70
Evaluation Completed

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

------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>laprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>531</td>
</tr>
<tr>
<td>readiden</td>
<td>80</td>
</tr>
<tr>
<td>readint</td>
<td>15</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>null</td>
<td>22</td>
</tr>
<tr>
<td>LE</td>
<td>10</td>
</tr>
</tbody>
</table>
------------------------
Total cells 748

Collate function (interpreted)
Enter Expression
< 2, 3, 3, 5, 6, 6, 7,
3, 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 suL> <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

68
| 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 | =val |
| 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 surprising in that the Pascal version is much faster.

Source Code (ELC)

<letrec fact
        <lambda <n>
                <if <<call <var eq> <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
*****************************************************************************
Total cells 327

fact (1)
Enter Expression
1
Evaluation Completed

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

Statistics
System time was 83 milliseconds
User time was 266 milliseconds

-------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcp</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>202</td>
</tr>
<tr>
<td>readimen</td>
<td>30</td>
</tr>
<tr>
<td>readint</td>
<td>4</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>equal</td>
<td>1</td>
</tr>
</tbody>
</table>
-------------------------------

Total cells 340

fact (2)
Enter Expression
2
Evaluation Completed
**Statistics**

System time was 133 milliseconds
User time was 233 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcpim</td>
<td>34</td>
</tr>
<tr>
<td>cons</td>
<td>222</td>
</tr>
<tr>
<td>raliden</td>
<td>30</td>
</tr>
<tr>
<td>readint</td>
<td>4</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>equal</td>
<td>3</td>
</tr>
<tr>
<td>suit</td>
<td>2</td>
</tr>
<tr>
<td>prod</td>
<td>2</td>
</tr>
</tbody>
</table>

---

Total cells 353

Fact (3)
Enter Expression
6
Evaluation Completed

**Statistics**

System time was 133 milliseconds
User time was 266 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcpim</td>
<td>34</td>
</tr>
<tr>
<td>cons</td>
<td>232</td>
</tr>
<tr>
<td>raliden</td>
<td>30</td>
</tr>
<tr>
<td>readint</td>
<td>4</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>equal</td>
<td>4</td>
</tr>
<tr>
<td>subt</td>
<td>3</td>
</tr>
<tr>
<td>prod</td>
<td>3</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>34</td>
</tr>
<tr>
<td>cons</td>
<td>242</td>
</tr>
<tr>
<td>readiden</td>
<td>30</td>
</tr>
<tr>
<td>readint</td>
<td>4</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>equal</td>
<td>5</td>
</tr>
<tr>
<td>subt</td>
<td>4</td>
</tr>
<tr>
<td>prod</td>
<td>4</td>
</tr>
</tbody>
</table>
```

Total cells 366

```
fact (4) Enter Expression
24
Evaluation Completed
```

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

```
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>34</td>
</tr>
<tr>
<td>cons</td>
<td>242</td>
</tr>
<tr>
<td>readiden</td>
<td>30</td>
</tr>
<tr>
<td>readint</td>
<td>4</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>equal</td>
<td>5</td>
</tr>
<tr>
<td>subt</td>
<td>4</td>
</tr>
<tr>
<td>prod</td>
<td>4</td>
</tr>
</tbody>
</table>

Total cells 379

```
fact (5) Enter Expression
120
Evaluation Completed
```

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

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcpri</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>252</td>
</tr>
<tr>
<td>readiden</td>
<td>30</td>
</tr>
<tr>
<td>readint</td>
<td>4</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>equal</td>
<td>6</td>
</tr>
<tr>
<td>sub</td>
<td>5</td>
</tr>
<tr>
<td>prod</td>
<td>5</td>
</tr>
</tbody>
</table>

Total cells 392

fact(6)
Enter Expression
720
Evaluation Completed

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

Statistics
System time was 116 milliseconds
User time was 416 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcpri</td>
<td>34</td>
</tr>
<tr>
<td>cons</td>
<td>262</td>
</tr>
<tr>
<td>readiden</td>
<td>30</td>
</tr>
<tr>
<td>readint</td>
<td>4</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>equal</td>
<td>7</td>
</tr>
</tbody>
</table>

97
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>272</td>
</tr>
<tr>
<td>readiden</td>
<td>30</td>
</tr>
<tr>
<td>readint</td>
<td>4</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>equal</td>
<td>8</td>
</tr>
<tr>
<td>subt</td>
<td>7</td>
</tr>
<tr>
<td>prod</td>
<td>7</td>
</tr>
</tbody>
</table>

Total cells 418

fact (8)
Enter Expression
40320
Evaluation Completed

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

Statistics
System time was 133 milliseconds
User time was 400 milliseconds

*****************************************************
User time was 433 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>282</td>
</tr>
<tr>
<td>readiden</td>
<td>30</td>
</tr>
<tr>
<td>readint</td>
<td>4</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>equal</td>
<td>9</td>
</tr>
<tr>
<td>subt</td>
<td>8</td>
</tr>
<tr>
<td>prod</td>
<td>8</td>
</tr>
</tbody>
</table>

Total cells 431

fact(9)
Enter Expression
362880
Evaluation Completed

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

Statistics

System time was 150 milliseconds
User time was 450 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>292</td>
</tr>
<tr>
<td>readiden</td>
<td>30</td>
</tr>
<tr>
<td>readint</td>
<td>4</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>equal</td>
<td>10</td>
</tr>
<tr>
<td>subt</td>
<td>9</td>
</tr>
<tr>
<td>prod</td>
<td>9</td>
</tr>
</tbody>
</table>
Total cells 444

fact(10)
Enter Expression 3623800
Evaluation Completed

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

System time was 133 milliseconds
User time was 450 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>jdcprim</td>
<td>34</td>
</tr>
<tr>
<td>cons</td>
<td>302</td>
</tr>
<tr>
<td>ireadiden</td>
<td>30</td>
</tr>
<tr>
<td>ireadint</td>
<td>4</td>
</tr>
<tr>
<td>iequal</td>
<td>6</td>
</tr>
<tr>
<td>islt</td>
<td>11</td>
</tr>
<tr>
<td>iprod</td>
<td>10</td>
</tr>
</tbody>
</table>

Total cells 457
Factorial function written in Berkeley 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
<table>
<thead>
<tr>
<th>Input n=2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>System Clock</td>
<td>33 millisec</td>
<td></td>
</tr>
<tr>
<td>User Clock</td>
<td>0 millisec</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input n=3</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>System Clock</td>
<td>33 millisec</td>
<td></td>
</tr>
<tr>
<td>User Clock</td>
<td>0 millisec</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input n=4</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>System Clock</td>
<td>33 millisec</td>
<td></td>
</tr>
<tr>
<td>User Clock</td>
<td>0 millisec</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input n=5</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>System Clock</td>
<td>33 millisec</td>
<td></td>
</tr>
<tr>
<td>User Clock</td>
<td>0 millisec</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input n=6</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>720</td>
<td></td>
</tr>
<tr>
<td>System Clock</td>
<td>33 millisec</td>
<td></td>
</tr>
<tr>
<td>User Clock</td>
<td>0 millisec</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input n=7</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5040</td>
<td></td>
</tr>
<tr>
<td>System Clock</td>
<td>16 millisec</td>
<td></td>
</tr>
<tr>
<td>User Clock</td>
<td>16 millisec</td>
<td></td>
</tr>
</tbody>
</table>
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)
Profiled Thu Dec 13 09:56 1984

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>func</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>printval</td>
</tr>
<tr>
<td>36</td>
<td>401</td>
<td>cellcount</td>
</tr>
<tr>
<td>62</td>
<td>11</td>
<td>equal</td>
</tr>
<tr>
<td>53</td>
<td>954</td>
<td>nullp</td>
</tr>
<tr>
<td>64</td>
<td>2064</td>
<td>first</td>
</tr>
<tr>
<td>94</td>
<td>1132</td>
<td>rest</td>
</tr>
<tr>
<td>122</td>
<td>302</td>
<td>cons</td>
</tr>
<tr>
<td>182</td>
<td>10</td>
<td>surt</td>
</tr>
<tr>
<td>208</td>
<td>10</td>
<td>prod</td>
</tr>
<tr>
<td>272</td>
<td>11</td>
<td>equalp</td>
</tr>
<tr>
<td>271</td>
<td>11</td>
<td>equal</td>
</tr>
<tr>
<td>533</td>
<td>150</td>
<td>atomp</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>547</td>
<td>954</td>
<td>nullp</td>
</tr>
<tr>
<td>599</td>
<td>73</td>
<td>assoc</td>
</tr>
<tr>
<td>643</td>
<td>22</td>
<td>pairlis</td>
</tr>
<tr>
<td>663</td>
<td>1</td>
<td>printval</td>
</tr>
<tr>
<td>702</td>
<td>56</td>
<td>readval</td>
</tr>
<tr>
<td>704</td>
<td>78</td>
<td>nonblank</td>
</tr>
<tr>
<td>714</td>
<td>22</td>
<td>readlist</td>
</tr>
<tr>
<td>741</td>
<td>0</td>
<td>readrea</td>
</tr>
<tr>
<td>770</td>
<td>4</td>
<td>readint</td>
</tr>
<tr>
<td>827</td>
<td>129</td>
<td>digit</td>
</tr>
<tr>
<td>835</td>
<td>163</td>
<td>letter</td>
</tr>
<tr>
<td>843</td>
<td>30</td>
<td>readident</td>
</tr>
<tr>
<td>867</td>
<td>56</td>
<td>readval</td>
</tr>
<tr>
<td>879</td>
<td>11</td>
<td>evcon</td>
</tr>
<tr>
<td>880</td>
<td>115</td>
<td>evils</td>
</tr>
<tr>
<td>881</td>
<td>42</td>
<td>apply</td>
</tr>
<tr>
<td>882</td>
<td>1</td>
<td>letrec</td>
</tr>
<tr>
<td>884</td>
<td>150</td>
<td>eval</td>
</tr>
<tr>
<td>947</td>
<td>1</td>
<td>letrec</td>
</tr>
<tr>
<td>1012</td>
<td>11</td>
<td>evcon</td>
</tr>
<tr>
<td>1041</td>
<td>115</td>
<td>evils</td>
</tr>
<tr>
<td>1093</td>
<td>31</td>
<td>applyprim</td>
</tr>
<tr>
<td>1198</td>
<td>42</td>
<td>apply</td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
<td>dcprim</td>
</tr>
<tr>
<td>1361</td>
<td>1</td>
<td>readfilename</td>
</tr>
</tbody>
</table>
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 is not surprising 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

```
<let rec reverse
    <lambda i>
    <if <<call <var i> i> <var 2>>
        <con >>>
```
<call <var cons> <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 sub> <var n> <con 1>>> <con 1>><call <var sub> <call <var fibo> <call <var sub> <var n> <con 1>>> <con 1>><call <var fibo> <call <var sub> <var n> <con 1>>> <con 2>>> <call <var fibo> <call <var sub> <var n> <con 1>>>>> <call <var reverse> <call <var fibo> <con 10>>>>>!

Results of fib(3..10) without a let statement
fibo(3)

Enter Expression

96
< 1, 1, 2>

Evaluation Completed

**********************************************************
Statistics
System time was 183 milliseconds
User time was 733 milliseconds
----------------------------------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>428</td>
</tr>
<tr>
<td>readiden</td>
<td>105</td>
</tr>
<tr>
<td>readint</td>
<td>11</td>
</tr>
<tr>
<td>letrec</td>
<td>12</td>
</tr>
<tr>
<td>equal</td>
<td>12</td>
</tr>
<tr>
<td>subt</td>
<td>3</td>
</tr>
<tr>
<td>sum</td>
<td>1</td>
</tr>
<tr>
<td>null</td>
<td>4</td>
</tr>
<tr>
<td>conr</td>
<td>2</td>
</tr>
</tbody>
</table>
----------------------------------------------------------
Total cells 662

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

Evaluation Completed

**********************************************************
Statistics
System time was 166 milliseconds
User time was 1250 milliseconds
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>570</td>
</tr>
<tr>
<td>readiden</td>
<td>105</td>
</tr>
<tr>
<td>readint</td>
<td>11</td>
</tr>
<tr>
<td>letrec</td>
<td>12</td>
</tr>
<tr>
<td>equal</td>
<td>39</td>
</tr>
<tr>
<td>succ</td>
<td>12</td>
</tr>
<tr>
<td>sum</td>
<td>4</td>
</tr>
<tr>
<td>null</td>
<td>5</td>
</tr>
<tr>
<td>contr</td>
<td>3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>976</td>
</tr>
<tr>
<td>readiden</td>
<td>105</td>
</tr>
<tr>
<td>readint</td>
<td>11</td>
</tr>
<tr>
<td>letrec</td>
<td>12</td>
</tr>
<tr>
<td>equal</td>
<td>120</td>
</tr>
<tr>
<td>succ</td>
<td>39</td>
</tr>
</tbody>
</table>

98
|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

---------------------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>----------</td>
<td>--------------</td>
</tr>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>2174</td>
</tr>
<tr>
<td>readiden</td>
<td>105</td>
</tr>
<tr>
<td>readint</td>
<td>11</td>
</tr>
<tr>
<td>letrec</td>
<td>12</td>
</tr>
<tr>
<td>equal</td>
<td>363</td>
</tr>
<tr>
<td>subt</td>
<td>120</td>
</tr>
<tr>
<td>sum</td>
<td>40</td>
</tr>
<tr>
<td>null</td>
<td>7</td>
</tr>
<tr>
<td>conr</td>
<td>5</td>
</tr>
</tbody>
</table>

---------------------------------------------
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

----------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcpmin</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>5748</td>
</tr>
<tr>
<td>readiden</td>
<td>105</td>
</tr>
<tr>
<td>readint</td>
<td>11</td>
</tr>
<tr>
<td>letrec</td>
<td>12</td>
</tr>
<tr>
<td>equal</td>
<td>1092</td>
</tr>
<tr>
<td>subt</td>
<td>363</td>
</tr>
<tr>
<td>sum</td>
<td>121</td>
</tr>
<tr>
<td>null</td>
<td>8</td>
</tr>
<tr>
<td>conv</td>
<td>6</td>
</tr>
</tbody>
</table>

----------------------------------
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

----------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
fib(9)

Enter Expression
< 1, 1, 2, 3, 5, 8, 13, 21, 34>

Evaluation Completed

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

Statistics
System time was 4650 milliseconds
User time was 154450 milliseconds

-------------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>48536</td>
</tr>
<tr>
<td>readiden</td>
<td>105</td>
</tr>
<tr>
<td>readint</td>
<td>11</td>
</tr>
<tr>
<td>letrec</td>
<td>12</td>
</tr>
<tr>
<td>equal</td>
<td>9840</td>
</tr>
<tr>
<td>subt</td>
<td>3279</td>
</tr>
<tr>
<td>sum</td>
<td>1093</td>
</tr>
<tr>
<td>null</td>
<td>10</td>
</tr>
</tbody>
</table>
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 cons>
      <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

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>---------------</td>
</tr>
<tr>
<td>ldcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>392</td>
</tr>
<tr>
<td>readiden</td>
<td>95</td>
</tr>
<tr>
<td>readint</td>
<td>9</td>
</tr>
<tr>
<td>letrec</td>
<td>12</td>
</tr>
<tr>
<td>equal</td>
<td>6</td>
</tr>
<tr>
<td>subt</td>
<td>1</td>
</tr>
<tr>
<td>sum</td>
<td>1</td>
</tr>
<tr>
<td>null</td>
<td>4</td>
</tr>
<tr>
<td>conr</td>
<td>2</td>
</tr>
</tbody>
</table>

Total cells 606

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

Evaluation Completed

************************************************************************
Statistics

System time was 183 milliseconds
User time was 783 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>----------------</td>
</tr>
<tr>
<td>jdcprim</td>
<td>84</td>
</tr>
<tr>
<td>icons</td>
<td>427</td>
</tr>
<tr>
<td>ireadiden</td>
<td>95</td>
</tr>
<tr>
<td>ireadint</td>
<td>9</td>
</tr>
<tr>
<td>jletrec</td>
<td>12</td>
</tr>
<tr>
<td>jequal</td>
<td>9</td>
</tr>
<tr>
<td>jsubt</td>
<td>2</td>
</tr>
<tr>
<td>jsum</td>
<td>2</td>
</tr>
<tr>
<td>jnull</td>
<td>5</td>
</tr>
<tr>
<td>jconr</td>
<td>3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>----------------</td>
</tr>
<tr>
<td>jdcprim</td>
<td>84</td>
</tr>
<tr>
<td>icons</td>
<td>462</td>
</tr>
<tr>
<td>ireadiden</td>
<td>95</td>
</tr>
</tbody>
</table>

105
| readint  | 9 | |
| letrc    | 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 |
-----------------------------
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>icprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>497</td>
</tr>
<tr>
<td>readiden</td>
<td>95</td>
</tr>
<tr>
<td>readint</td>
<td>9</td>
</tr>
<tr>
<td>letrc</td>
<td>12</td>
</tr>
<tr>
<td>equal</td>
<td>15</td>
</tr>
<tr>
<td>subt</td>
<td>4</td>
</tr>
<tr>
<td>sum</td>
<td>4</td>
</tr>
<tr>
<td>null</td>
<td>7</td>
</tr>
<tr>
<td>conr</td>
<td>5</td>
</tr>
</tbody>
</table>

-----------------------------
| 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

-----------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcpri</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>532</td>
</tr>
<tr>
<td>readiden</td>
<td>95</td>
</tr>
<tr>
<td>readint</td>
<td>9</td>
</tr>
<tr>
<td>letrec</td>
<td>12</td>
</tr>
<tr>
<td>equal</td>
<td>18</td>
</tr>
<tr>
<td>subt</td>
<td>5</td>
</tr>
<tr>
<td>sum</td>
<td>5</td>
</tr>
<tr>
<td>null</td>
<td>8</td>
</tr>
<tr>
<td>conr</td>
<td>6</td>
</tr>
</tbody>
</table>
-----------------------------
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
Total cells 1218

Results of restriction (interpreted)
Enter Expression
<< 6, 5>
, < 3, 9>
>
Evaluation Completed

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

Statistics
System time was 516 milliseconds
User time was 17966 milliseconds

----------------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>dcprim</td>
<td>34</td>
</tr>
<tr>
<td>cons</td>
<td>837</td>
</tr>
<tr>
<td>readiden</td>
<td>220</td>
</tr>
<tr>
<td>readint</td>
<td>8</td>
</tr>
<tr>
<td>letrec</td>
<td>24</td>
</tr>
<tr>
<td>eval</td>
<td>1</td>
</tr>
<tr>
<td>equal</td>
<td>12</td>
</tr>
<tr>
<td>null</td>
<td>29</td>
</tr>
<tr>
<td>len</td>
<td>3</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
</tr>
</tbody>
</table>

----------------------------------------
| Total cells | 1218          |

Profile for restriction program
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Results of restriction function (Compiled)
Enter Expression

```
<< 6, 5, 8, 9 >>
```

Evaluation Completed

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

-------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcpri 84</td>
<td></td>
</tr>
<tr>
<td>cons 837</td>
<td></td>
</tr>
<tr>
<td>readiden 220</td>
<td></td>
</tr>
<tr>
<td>readint 8</td>
<td></td>
</tr>
<tr>
<td>letrec 24</td>
<td></td>
</tr>
<tr>
<td>eval 1</td>
<td></td>
</tr>
<tr>
<td>equal 12</td>
<td></td>
</tr>
<tr>
<td>null 29</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td></td>
</tr>
</tbody>
</table>

---
<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 isinfunc>
    <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 noinfnc>>>>
<letrec restric
    <lambda <T k>
        <if <<call <var equal>
            <call <var isinfunc> <var T>>
        <con true>>
        <if <<call <var null> <var T>>
            <con <>>
        <if <<call <var eual>
            <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>>
<table>
<thead>
<tr>
<th>Number</th>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>880</td>
<td>606</td>
<td>evlis</td>
</tr>
<tr>
<td>881</td>
<td>202</td>
<td>apply</td>
</tr>
<tr>
<td>882</td>
<td>1</td>
<td>letrec</td>
</tr>
<tr>
<td>884</td>
<td>761</td>
<td>eval</td>
</tr>
<tr>
<td>947</td>
<td>1</td>
<td>letrec</td>
</tr>
<tr>
<td>1012</td>
<td>51</td>
<td>evcon</td>
</tr>
<tr>
<td>1041</td>
<td>606</td>
<td>evlis</td>
</tr>
<tr>
<td>1093</td>
<td>151</td>
<td>applyprim</td>
</tr>
<tr>
<td>1198</td>
<td>202</td>
<td>apply</td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
<td>dcprim</td>
</tr>
<tr>
<td>1261</td>
<td>1</td>
<td>readfname</td>
</tr>
<tr>
<td>Line</td>
<td>Count</td>
<td>func</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>--------------</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>51</td>
<td>printval</td>
</tr>
<tr>
<td>36</td>
<td>1136</td>
<td>cellcount</td>
</tr>
<tr>
<td>63</td>
<td>5487</td>
<td>nullp</td>
</tr>
<tr>
<td>84</td>
<td>11793</td>
<td>first</td>
</tr>
<tr>
<td>94</td>
<td>6357</td>
<td>rest</td>
</tr>
<tr>
<td>122</td>
<td>963</td>
<td>cons</td>
</tr>
<tr>
<td>158</td>
<td>50</td>
<td>sum</td>
</tr>
<tr>
<td>330</td>
<td>51</td>
<td>GTp</td>
</tr>
<tr>
<td>340</td>
<td>51</td>
<td>GT</td>
</tr>
<tr>
<td>539</td>
<td>761</td>
<td>atomp</td>
</tr>
<tr>
<td>547</td>
<td>5487</td>
<td>nullp</td>
</tr>
<tr>
<td>599</td>
<td>454</td>
<td>assoc</td>
</tr>
<tr>
<td>643</td>
<td>153</td>
<td>pairlis</td>
</tr>
<tr>
<td>663</td>
<td>51</td>
<td>printval</td>
</tr>
<tr>
<td>702</td>
<td>63</td>
<td>readval</td>
</tr>
<tr>
<td>704</td>
<td>88</td>
<td>nonblank</td>
</tr>
<tr>
<td>714</td>
<td>25</td>
<td>readlist</td>
</tr>
<tr>
<td>770</td>
<td>3</td>
<td>readint</td>
</tr>
<tr>
<td>827</td>
<td>151</td>
<td>digit</td>
</tr>
<tr>
<td>835</td>
<td>189</td>
<td>letter</td>
</tr>
<tr>
<td>843</td>
<td>35</td>
<td>readident</td>
</tr>
<tr>
<td>867</td>
<td>63</td>
<td>readval</td>
</tr>
<tr>
<td>879</td>
<td>51</td>
<td>evcon</td>
</tr>
</tbody>
</table>

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

Statistics
System time was 266 milliseconds
User time was 1716 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>963</td>
</tr>
<tr>
<td>readiden</td>
<td>35</td>
</tr>
<tr>
<td>readint</td>
<td>3</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>GT</td>
<td>51</td>
</tr>
<tr>
<td>sum</td>
<td>50</td>
</tr>
</tbody>
</table>

Total cells 1192

Results of interval (interpreted)

Statistics
System time was 616 milliseconds
User time was 19966 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>963</td>
</tr>
<tr>
<td>readiden</td>
<td>35</td>
</tr>
<tr>
<td>readint</td>
<td>3</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>GT</td>
<td>51</td>
</tr>
</tbody>
</table>
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>
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>539</td>
<td>466</td>
</tr>
<tr>
<td>547</td>
<td>3862</td>
</tr>
<tr>
<td>564</td>
<td>11</td>
</tr>
<tr>
<td>599</td>
<td>226</td>
</tr>
<tr>
<td>643</td>
<td>56</td>
</tr>
<tr>
<td>663</td>
<td>11</td>
</tr>
<tr>
<td>702</td>
<td>180</td>
</tr>
<tr>
<td>704</td>
<td>256</td>
</tr>
<tr>
<td>714</td>
<td>76</td>
</tr>
<tr>
<td>770</td>
<td>9</td>
</tr>
<tr>
<td>827</td>
<td>410</td>
</tr>
<tr>
<td>835</td>
<td>514</td>
</tr>
<tr>
<td>843</td>
<td>95</td>
</tr>
<tr>
<td>867</td>
<td>180</td>
</tr>
<tr>
<td>879</td>
<td>38</td>
</tr>
<tr>
<td>880</td>
<td>375</td>
</tr>
<tr>
<td>881</td>
<td>136</td>
</tr>
<tr>
<td>882</td>
<td>2</td>
</tr>
<tr>
<td>884</td>
<td>466</td>
</tr>
<tr>
<td>947</td>
<td>2</td>
</tr>
<tr>
<td>1012</td>
<td>38</td>
</tr>
<tr>
<td>1041</td>
<td>375</td>
</tr>
<tr>
<td>1093</td>
<td>116</td>
</tr>
<tr>
<td>1198</td>
<td>136</td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
</tr>
<tr>
<td>1261</td>
<td>1</td>
</tr>
</tbody>
</table>
Use: time 0 millisec

fib(2),
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 ELI C fibonacci sequence generating functions
Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Wed Dec 12 12:48 1984 test11.r
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
equal

112
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

110
\[
\begin{array}{c|c|c}
|   | \text{equal} & 24 \\
|   | \text{subt}  & 7  \\
|   | \text{sum}   & 7  \\
|   | \text{null}  & 10 \\
|   | \text{conr}  & 8  \\
\end{array}
\]

---

Total cells 858

\[
\text{fib}(10)
\]

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

----------

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>--------</td>
<td>---------------</td>
</tr>
<tr>
<td>dcrpr</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>637</td>
</tr>
<tr>
<td>readidn</td>
<td>95</td>
</tr>
<tr>
<td>readint</td>
<td>9</td>
</tr>
<tr>
<td>letrec</td>
<td>12</td>
</tr>
<tr>
<td>equal</td>
<td>27</td>
</tr>
<tr>
<td>subt</td>
<td>8</td>
</tr>
<tr>
<td>sum</td>
<td>8</td>
</tr>
<tr>
<td>null</td>
<td>11</td>
</tr>
<tr>
<td>conr</td>
<td>9</td>
</tr>
</tbody>
</table>

----------

Total cells 900

109
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprom</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>567</td>
</tr>
<tr>
<td>readiden</td>
<td>95</td>
</tr>
<tr>
<td>readint</td>
<td>9</td>
</tr>
<tr>
<td>letrec</td>
<td>12</td>
</tr>
<tr>
<td>equal</td>
<td>21</td>
</tr>
<tr>
<td>sub</td>
<td>6</td>
</tr>
<tr>
<td>sum</td>
<td>6</td>
</tr>
<tr>
<td>null</td>
<td>9</td>
</tr>
<tr>
<td>conr</td>
<td>7</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprom</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>602</td>
</tr>
<tr>
<td>readiden</td>
<td>95</td>
</tr>
<tr>
<td>readint</td>
<td>9</td>
</tr>
<tr>
<td>letrec</td>
<td>12</td>
</tr>
</tbody>
</table>

103
<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>func</td>
</tr>
<tr>
<td>33</td>
<td>7</td>
<td>printval</td>
</tr>
<tr>
<td>36</td>
<td>1162</td>
<td>cellcount</td>
</tr>
<tr>
<td>62</td>
<td>12</td>
<td>equal</td>
</tr>
<tr>
<td>63</td>
<td>4184</td>
<td>nullp</td>
</tr>
<tr>
<td>84</td>
<td>8859</td>
<td>first</td>
</tr>
<tr>
<td>94</td>
<td>4699</td>
<td>rest</td>
</tr>
<tr>
<td>122</td>
<td>837</td>
<td>cons</td>
</tr>
<tr>
<td>273</td>
<td>12</td>
<td>equalp</td>
</tr>
<tr>
<td>291</td>
<td>12</td>
<td>equal</td>
</tr>
<tr>
<td>539</td>
<td>441</td>
<td>atomp</td>
</tr>
<tr>
<td>547</td>
<td>4184</td>
<td>nullp</td>
</tr>
<tr>
<td>564</td>
<td>29</td>
<td>null</td>
</tr>
<tr>
<td>576</td>
<td>9</td>
<td>lenp</td>
</tr>
<tr>
<td>587</td>
<td>3</td>
<td>len</td>
</tr>
<tr>
<td>599</td>
<td>228</td>
<td>assoc</td>
</tr>
<tr>
<td>643</td>
<td>69</td>
<td>pairlis</td>
</tr>
<tr>
<td>663</td>
<td>7</td>
<td>printval</td>
</tr>
<tr>
<td>702</td>
<td>387</td>
<td>readval</td>
</tr>
<tr>
<td>704</td>
<td>546</td>
<td>nonblank</td>
</tr>
<tr>
<td>714</td>
<td>159</td>
<td>readlist</td>
</tr>
<tr>
<td>770</td>
<td>8</td>
<td>readint</td>
</tr>
<tr>
<td>827</td>
<td>1011</td>
<td>digit</td>
</tr>
<tr>
<td>835</td>
<td>1239</td>
<td>letter</td>
</tr>
<tr>
<td>843</td>
<td>220</td>
<td>readident</td>
</tr>
<tr>
<td>879</td>
<td>41</td>
<td>evcon</td>
</tr>
<tr>
<td>880</td>
<td>325</td>
<td>evlis</td>
</tr>
<tr>
<td>881</td>
<td>145</td>
<td>apply</td>
</tr>
<tr>
<td>882</td>
<td>4</td>
<td>letrec</td>
</tr>
<tr>
<td>884</td>
<td>441</td>
<td>eval</td>
</tr>
<tr>
<td>947</td>
<td>4</td>
<td>letrec</td>
</tr>
<tr>
<td>1012</td>
<td>41</td>
<td>evcon</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>1041</td>
<td>325</td>
<td>evlis</td>
</tr>
<tr>
<td>1093</td>
<td>115</td>
<td>applyprim</td>
</tr>
<tr>
<td>1198</td>
<td>145</td>
<td>apply</td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
<td>dcprim</td>
</tr>
<tr>
<td>1261</td>
<td>1</td>
<td>readfname</td>
</tr>
</tbody>
</table>
Vector Product Function

Purpose
Vector Product 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 <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>>
               124
Results of vectorprd function (Compiled)

Enter Expression
< 10, 160, 28, 27>

Evaluation Completed

*******************************************************************************
Statistics
System time was 216 milliseconds
User time was 1593 milliseconds

--------------------------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcpriw</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>720</td>
</tr>
<tr>
<td>readiden</td>
<td>150</td>
</tr>
<tr>
<td>readint</td>
<td>10</td>
</tr>
</tbody>
</table>
--------------------------------------------------
125
Results of vectorproduct function (Interpreted)
Enter Expression
< 10, 160, 28, 27>
Evaluation Completed

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

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>720</td>
</tr>
<tr>
<td>readilen</td>
<td>150</td>
</tr>
<tr>
<td>readint</td>
<td>10</td>
</tr>
<tr>
<td>letrec</td>
<td>18</td>
</tr>
<tr>
<td>eval</td>
<td>5</td>
</tr>
<tr>
<td>len</td>
<td>10</td>
</tr>
<tr>
<td>equal</td>
<td>5</td>
</tr>
<tr>
<td>null</td>
<td>23</td>
</tr>
<tr>
<td>prod</td>
<td>8</td>
</tr>
</tbody>
</table>

Total cells 1033
Profile for vectorproduct function

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>977</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>3579</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>7535</td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>4038</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>720</td>
<td></td>
</tr>
<tr>
<td>208</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>258</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>273</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>291</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>539</td>
<td>395</td>
<td></td>
</tr>
<tr>
<td>547</td>
<td>3579</td>
<td></td>
</tr>
<tr>
<td>564</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>576</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>587</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>599</td>
<td>202</td>
<td></td>
</tr>
<tr>
<td>643</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>663</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>702</td>
<td>272</td>
<td></td>
</tr>
<tr>
<td>704</td>
<td>384</td>
<td></td>
</tr>
<tr>
<td>714</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>770</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>827</td>
<td>659</td>
<td></td>
</tr>
<tr>
<td>835</td>
<td>819</td>
<td></td>
</tr>
<tr>
<td>843</td>
<td>150</td>
<td></td>
</tr>
</tbody>
</table>

127
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>867</td>
<td>272</td>
<td>readval</td>
<td></td>
</tr>
<tr>
<td>879</td>
<td>28</td>
<td>evcon</td>
<td></td>
</tr>
<tr>
<td>880</td>
<td>310</td>
<td>evlis</td>
<td></td>
</tr>
<tr>
<td>881</td>
<td>129</td>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>882</td>
<td>3</td>
<td>letrec</td>
<td></td>
</tr>
<tr>
<td>884</td>
<td>395</td>
<td>eval</td>
<td></td>
</tr>
<tr>
<td>947</td>
<td>3</td>
<td>letrec</td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td>28</td>
<td>evcon</td>
<td></td>
</tr>
<tr>
<td>1041</td>
<td>310</td>
<td>evlis</td>
<td></td>
</tr>
<tr>
<td>1093</td>
<td>102</td>
<td>applyprim</td>
<td></td>
</tr>
<tr>
<td>1193</td>
<td>129</td>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
<td>dcprim</td>
<td></td>
</tr>
<tr>
<td>1261</td>
<td>1</td>
<td>readfname</td>
<td></td>
</tr>
</tbody>
</table>
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 ccns>
            <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 bool>
            <var arg>>
            <call <var rest> <var L>>>>>>>>
        <call
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

----------------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldcprim</td>
<td>94</td>
</tr>
<tr>
<td>cons</td>
<td>562</td>
</tr>
<tr>
<td>readiden</td>
<td>79</td>
</tr>
<tr>
<td>readint</td>
<td>13</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>eval</td>
<td>11</td>
</tr>
<tr>
<td>null</td>
<td>11</td>
</tr>
<tr>
<td>GE</td>
<td>10</td>
</tr>
</tbody>
</table>
----------------------------------------
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

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>562</td>
</tr>
<tr>
<td>readiden</td>
<td>79</td>
</tr>
<tr>
<td>readint</td>
<td>13</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>eval</td>
<td>11</td>
</tr>
<tr>
<td>null</td>
<td>11</td>
</tr>
<tr>
<td>GE</td>
<td>10</td>
</tr>
</tbody>
</table>

| Total cells | 776 |

### Profile for the filtering function

**Berkeley Pascal PXP -- Version 2.12 (5/11/83)**

**Profiled Thu Dec 13 13:40 1984**

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>8</td>
<td>printval</td>
</tr>
<tr>
<td>36</td>
<td>720</td>
<td>ceilcount</td>
</tr>
<tr>
<td>63</td>
<td>1838</td>
<td>nullp</td>
</tr>
<tr>
<td>84</td>
<td>4010</td>
<td>first</td>
</tr>
<tr>
<td>94</td>
<td>2185</td>
<td>rest</td>
</tr>
<tr>
<td>122</td>
<td>562</td>
<td>cons</td>
</tr>
<tr>
<td>258</td>
<td>17</td>
<td>sub</td>
</tr>
<tr>
<td>357</td>
<td>10</td>
<td>GEp</td>
</tr>
<tr>
<td>367</td>
<td>10</td>
<td>GE</td>
</tr>
<tr>
<td>539</td>
<td>275</td>
<td>atomp</td>
</tr>
</tbody>
</table>

131
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>547</td>
<td>1838</td>
<td>nullp</td>
<td></td>
</tr>
<tr>
<td>564</td>
<td>11</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>599</td>
<td>135</td>
<td>assoc</td>
<td></td>
</tr>
<tr>
<td>643</td>
<td>55</td>
<td>pairlis</td>
<td></td>
</tr>
<tr>
<td>663</td>
<td>9</td>
<td>printval</td>
<td></td>
</tr>
<tr>
<td>702</td>
<td>151</td>
<td>readval</td>
<td></td>
</tr>
<tr>
<td>704</td>
<td>210</td>
<td>nonblank</td>
<td></td>
</tr>
<tr>
<td>714</td>
<td>59</td>
<td>readlist</td>
<td></td>
</tr>
<tr>
<td>770</td>
<td>13</td>
<td>readint</td>
<td></td>
</tr>
<tr>
<td>827</td>
<td>332</td>
<td>digit</td>
<td></td>
</tr>
<tr>
<td>835</td>
<td>424</td>
<td>letter</td>
<td></td>
</tr>
<tr>
<td>843</td>
<td>79</td>
<td>readident</td>
<td></td>
</tr>
<tr>
<td>867</td>
<td>151</td>
<td>readval</td>
<td></td>
</tr>
<tr>
<td>879</td>
<td>21</td>
<td>evcon</td>
<td></td>
</tr>
<tr>
<td>880</td>
<td>210</td>
<td>evlis</td>
<td></td>
</tr>
<tr>
<td>881</td>
<td>77</td>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>882</td>
<td>1</td>
<td>letrec</td>
<td></td>
</tr>
<tr>
<td>884</td>
<td>275</td>
<td>eval</td>
<td></td>
</tr>
<tr>
<td>947</td>
<td>1</td>
<td>letrec</td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td>21</td>
<td>evcon</td>
<td></td>
</tr>
<tr>
<td>1041</td>
<td>210</td>
<td>evlis</td>
<td></td>
</tr>
<tr>
<td>1093</td>
<td>55</td>
<td>applyprim</td>
<td></td>
</tr>
<tr>
<td>1198</td>
<td>77</td>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
<td>dcprim</td>
<td></td>
</tr>
<tr>
<td>1261</td>
<td>1</td>
<td>readfname</td>
<td></td>
</tr>
</tbody>
</table>
Periodic Sequence Generator

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

Source Code
<letrec reverse
  <l ambda <L>
    <if <<call <var null> <var L>>
      <con <>>
    <call <var cons>
      <call <var reverse>
        <call <var rest> <var L>>>
      <call <var sub> <var L> <con 1>>>>>>>
  </letrec fibo
  <l ambda <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 $x_1 = 2$, $x_2 = 9$, and $x_k = (x_{k-1}) - (x_{k-2})$ for $k = 3, 4, 5, \ldots$

Enter Expression

\[
\begin{align*}
< & \quad 2, \quad 9, \quad 7, \quad -2, \quad -9, \quad -7, \\
& \quad 2, \quad 9, \quad 7, \quad -2, \quad -9, \quad -7, \\
& \quad 2, \quad 9, \quad 7, \quad -2, \quad -9, \quad -7, \\
& \quad 2, \quad 9, \quad 7, \quad -2, \quad -9, \quad -7>
\end{align*}
\]

Evaluation Completed

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

Statistics

System time was 283 milliseconds
User time was 2916 milliseconds

------------------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcpim</td>
<td>84</td>
</tr>
<tr>
<td>iconso</td>
<td>1064</td>
</tr>
<tr>
<td>readiden</td>
<td>87</td>
</tr>
<tr>
<td>readint</td>
<td>8</td>
</tr>
<tr>
<td>letrec</td>
<td>12</td>
</tr>
<tr>
<td>equal</td>
<td>46</td>
</tr>
<tr>
<td>subt</td>
<td>44</td>
</tr>
<tr>
<td>null</td>
<td>25</td>
</tr>
<tr>
<td>conv</td>
<td>23</td>
</tr>
</tbody>
</table>

------------------------------------------
Total cells 1393

Profile of seq2 (periodic function)

Berkeley Pascal PXP -- Version 2.12 (5/11/83)

Profiled Thu Dec 13 10:10 1984

134
<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>21</td>
</tr>
<tr>
<td>36</td>
<td>1131</td>
</tr>
<tr>
<td>62</td>
<td>38</td>
</tr>
<tr>
<td>65</td>
<td>7652</td>
</tr>
<tr>
<td>84</td>
<td>16163</td>
</tr>
<tr>
<td>94</td>
<td>8626</td>
</tr>
<tr>
<td>122</td>
<td>932</td>
</tr>
<tr>
<td>137</td>
<td>20</td>
</tr>
<tr>
<td>183</td>
<td>36</td>
</tr>
<tr>
<td>258</td>
<td>20</td>
</tr>
<tr>
<td>273</td>
<td>38</td>
</tr>
<tr>
<td>291</td>
<td>38</td>
</tr>
<tr>
<td>539</td>
<td>881</td>
</tr>
<tr>
<td>547</td>
<td>7652</td>
</tr>
<tr>
<td>564</td>
<td>21</td>
</tr>
<tr>
<td>593</td>
<td>438</td>
</tr>
<tr>
<td>643</td>
<td>116</td>
</tr>
<tr>
<td>663</td>
<td>21</td>
</tr>
<tr>
<td>702</td>
<td>163</td>
</tr>
<tr>
<td>704</td>
<td>231</td>
</tr>
<tr>
<td>714</td>
<td>68</td>
</tr>
<tr>
<td>770</td>
<td>8</td>
</tr>
<tr>
<td>827</td>
<td>379</td>
</tr>
<tr>
<td>835</td>
<td>474</td>
</tr>
<tr>
<td>843</td>
<td>87</td>
</tr>
<tr>
<td>867</td>
<td>163</td>
</tr>
<tr>
<td>879</td>
<td>59</td>
</tr>
<tr>
<td>980</td>
<td>736</td>
</tr>
<tr>
<td>881</td>
<td>267</td>
</tr>
<tr>
<td>882</td>
<td>2</td>
</tr>
<tr>
<td>884</td>
<td>881</td>
</tr>
<tr>
<td>947</td>
<td>2</td>
</tr>
</tbody>
</table>

```
func
printval
cellicount
equal
nullp
first
rest
cons
conr
subt
sub
equalp
equal
atomp
nullp
null
assoc
pairlis
printval
readval
nonblank
readlist
readint
digit
letter
readident
readval
evcon
evlis
apply
letrec
eval
letrec
```
<table>
<thead>
<tr>
<th>X</th>
<th>Y1</th>
<th>Y2</th>
<th>Y3</th>
<th>Y4</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>0.694658</td>
<td>0.719340</td>
<td>0.965689</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>0.707107</td>
<td>0.707107</td>
<td>1.000000</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>0.719340</td>
<td>0.694658</td>
<td>1.035330</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>0.731354</td>
<td>0.681998</td>
<td>1.072369</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>0.743145</td>
<td>0.669131</td>
<td>1.110613</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>0.754710</td>
<td>0.656059</td>
<td>1.150368</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>0.766044</td>
<td>0.642788</td>
<td>1.191754</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>0.777146</td>
<td>0.629320</td>
<td>1.234397</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>0.788011</td>
<td>0.615661</td>
<td>1.279942</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>0.798636</td>
<td>0.601815</td>
<td>1.327045</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>0.809017</td>
<td>0.587735</td>
<td>1.376332</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>0.819152</td>
<td>0.573576</td>
<td>1.428148</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>0.829038</td>
<td>0.559193</td>
<td>1.482561</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>0.838671</td>
<td>0.544639</td>
<td>1.539865</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>0.848048</td>
<td>0.529919</td>
<td>1.600335</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>0.857167</td>
<td>0.515038</td>
<td>1.664279</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>0.866025</td>
<td>0.500000</td>
<td>1.732051</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>0.874620</td>
<td>0.484810</td>
<td>1.804048</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>0.882948</td>
<td>0.469472</td>
<td>1.880726</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>0.891007</td>
<td>0.453990</td>
<td>1.962611</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>0.896794</td>
<td>0.438371</td>
<td>2.050304</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>0.906308</td>
<td>0.422618</td>
<td>2.145027</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>0.913545</td>
<td>0.406737</td>
<td>2.246073</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>0.920505</td>
<td>0.390731</td>
<td>2.355852</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>0.927184</td>
<td>0.374607</td>
<td>2.475087</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>0.933580</td>
<td>0.358368</td>
<td>2.605033</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>0.939693</td>
<td>0.342020</td>
<td>2.747477</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>0.945519</td>
<td>0.325568</td>
<td>2.904211</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>0.951057</td>
<td>0.309017</td>
<td>3.077848</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>0.956305</td>
<td>0.292372</td>
<td>3.270953</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>0.961262</td>
<td>0.275637</td>
<td>3.487414</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>0.965326</td>
<td>0.258819</td>
<td>3.732051</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>0.970296</td>
<td>0.241922</td>
<td>4.010781</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>0.974370</td>
<td>0.224951</td>
<td>4.331476</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>0.978148</td>
<td>0.207912</td>
<td>4.704030</td>
<td></td>
</tr>
</tbody>
</table>
< 9, 0.156434, 0.987688, 0.158334>
< 10, 0.173648, 0.934608, 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, 0.404326>
< 23, 0.390731, 0.920505, 0.424475>
< 24, 0.406737, 0.913545, 0.445229>
< 25, 0.422618, 0.906308, 0.466338>
< 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, 0.857107, 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, 0.700203>
< 36, 0.587785, 0.809017, 0.726543>
< 37, 0.601815, 0.798636, 0.753554>
< 38, 0.615661, 0.788011, 0.781286>
< 39, 0.629320, 0.777140, 0.809734>
< 40, 0.642788, 0.766044, 0.839100>
< 41, 0.656059, 0.754710, 0.869287>
< 42, 0.669131, 0.743145, 0.899404>
< 43, 0.681998, 0.731334, 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 <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, 0.997846, 0.069927
, 5, 0.087156, 0.997036, 0.087389
, 6, 0.104528, 0.996203, 0.105104
, 7, 0.121869, 0.995337, 0.122785
, 8, 0.139173, 0.994444, 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, \( \text{par} \), is also used. Map takes one function and applies it to all the elements of a list, where \( \text{par} \) takes a list of functions and applies each one to the same argument. It is clear that mapping the \( \text{par} \) 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**

```plaintext
<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>>>>>
```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>947</td>
<td>1</td>
<td>letrec</td>
</tr>
<tr>
<td>1012</td>
<td>6</td>
<td>evcon</td>
</tr>
<tr>
<td>1041</td>
<td>169</td>
<td>evlis</td>
</tr>
<tr>
<td>1093</td>
<td>45</td>
<td>applyprim</td>
</tr>
<tr>
<td>1198</td>
<td>52</td>
<td>apply</td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
<td>dcprim</td>
</tr>
<tr>
<td>1261</td>
<td>1</td>
<td>readfname</td>
</tr>
<tr>
<td>Line</td>
<td>Count</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>614</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>1656</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>3439</td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>1861</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>472</td>
<td></td>
</tr>
<tr>
<td>183</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>233</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>273</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>291</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>539</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>547</td>
<td>1656</td>
<td></td>
</tr>
<tr>
<td>576</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>587</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>599</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>643</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>663</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>702</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>704</td>
<td>218</td>
<td></td>
</tr>
<tr>
<td>714</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>770</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>827</td>
<td>375</td>
<td></td>
</tr>
<tr>
<td>835</td>
<td>469</td>
<td></td>
</tr>
<tr>
<td>843</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>867</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>873</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>880</td>
<td>169</td>
<td></td>
</tr>
<tr>
<td>882</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>884</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

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

<table>
<thead>
<tr>
<th>Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>System time was</td>
</tr>
<tr>
<td>User time was</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>472</td>
</tr>
<tr>
<td>readiden</td>
<td>91</td>
</tr>
<tr>
<td>readint</td>
<td>3</td>
</tr>
<tr>
<td>letrec</td>
<td>6</td>
</tr>
<tr>
<td>eval</td>
<td>1</td>
</tr>
<tr>
<td>len</td>
<td>1</td>
</tr>
<tr>
<td>divi</td>
<td>1</td>
</tr>
<tr>
<td>equal</td>
<td>6</td>
</tr>
<tr>
<td>subt</td>
<td>5</td>
</tr>
</tbody>
</table>

| 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.f
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>>
        <call <var cons>
          <call <var cons>
            <call <var first> <var L>>
            <call <var first> <var r>>>>
          <call <var rest> <var r>>>>>>>
        <let <<split>
Evaluation Completed

******************** Statistics ***********************

System time was 633 milliseconds
User time was 24433 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcpprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>907</td>
</tr>
<tr>
<td>readiden</td>
<td>170</td>
</tr>
<tr>
<td>readint</td>
<td>9</td>
</tr>
<tr>
<td>leterr</td>
<td>18</td>
</tr>
<tr>
<td>null</td>
<td>46</td>
</tr>
<tr>
<td>len</td>
<td>6</td>
</tr>
<tr>
<td>equal</td>
<td>13</td>
</tr>
<tr>
<td>memt</td>
<td>6</td>
</tr>
</tbody>
</table>

Total cells 1259

Profile for overlay program
Perkel Pascal PXP -- Version 2.12 (5/11/83)
Profiled Thu Dec 13 14:05 1984

<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>13</td>
<td>printval</td>
</tr>
<tr>
<td>36</td>
<td>1203</td>
<td>cellcount</td>
</tr>
<tr>
<td>62</td>
<td>13</td>
<td>equal</td>
</tr>
<tr>
<td>63</td>
<td>6758</td>
<td>null</td>
</tr>
<tr>
<td>84</td>
<td>14295</td>
<td>first</td>
</tr>
<tr>
<td>94</td>
<td>7556</td>
<td>rest</td>
</tr>
</tbody>
</table>
Evaluaticn Completed

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

Statistics
System time was 193 milliseconds
User time was 2333 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>907</td>
</tr>
<tr>
<td>readiden</td>
<td>170</td>
</tr>
<tr>
<td>readint</td>
<td>9</td>
</tr>
<tr>
<td>letrec</td>
<td>18</td>
</tr>
<tr>
<td>null</td>
<td>46</td>
</tr>
<tr>
<td>len</td>
<td>6</td>
</tr>
<tr>
<td>equal</td>
<td>13</td>
</tr>
<tr>
<td>memb</td>
<td>6</td>
</tr>
</tbody>
</table>

Total cells 1259

Results of overlay (Interpreted)
Enter Expression
<< 3, 4>
, < 6, 5>
, < 8, 9>
, < 7, 2>
>
Enter Expression to the table. (Compiled)

Results of overlay function adding the value 3 to the table.

Letrec overlay

\[ \text{letrec } \text{overlay} \Rightarrow \lambda \text{pr} \Rightarrow \begin{cases} \text{true} & \text{if equal}(\text{isfinfunc}(\text{pr}), \text{true}) \\ \text{null}(\text{pr}) & \text{if null}(\text{first}(\text{pr})) \\ \text{cons}(\text{first}(\text{pr}), \text{overlay}(\text{first}(\text{pr}), \text{rest}(\text{pr}))) & \text{else} \end{cases} \]

\text{letrec repr } \Rightarrow \begin{cases} \text{finset} \Rightarrow \{1, 6, 8\} & \text{true} \\ \text{null} & \text{false} \end{cases} \]

\text{letrec equal } \Rightarrow \begin{cases} \text{true} & \text{if equal}(\text{finset}, \{3\}) \\ \text{false} & \text{else} \end{cases}

\text{letrec isfinfunc } \Rightarrow \begin{cases} \text{true} & \text{if equal}(\text{finset}, \{1, 6, 8\}) \\ \text{false} & \text{else} \end{cases}

\text{letrec overlay } \Rightarrow \begin{cases} \text{true} & \text{if equal}(\text{isfinfunc}, \text{true}) \\ \text{null}(\text{pr}) & \text{if null}(\text{first}(\text{pr})) \\ \text{cons}(\text{first}(\text{pr}), \text{overlay}(\text{first}(\text{pr}), \text{rest}(\text{pr}))) & \text{else} \end{cases} \]

\text{letrec repr } \Rightarrow \begin{cases} \text{finset} \Rightarrow \{1, 6, 8\} & \text{true} \\ \text{null} & \text{false} \end{cases} \]

\text{letrec equal } \Rightarrow \begin{cases} \text{true} & \text{if equal}(\text{finset}, \{3\}) \\ \text{false} & \text{else} \end{cases}

\text{letrec isfinfunc } \Rightarrow \begin{cases} \text{true} & \text{if equal}(\text{finset}, \{1, 6, 8\}) \\ \text{false} & \text{else} \end{cases}
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>

137
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1012</td>
<td>59</td>
<td>evcon</td>
<td></td>
</tr>
<tr>
<td>1041</td>
<td>738</td>
<td>evlis</td>
<td></td>
</tr>
<tr>
<td>1093</td>
<td>227</td>
<td>applyprim</td>
<td></td>
</tr>
<tr>
<td>1198</td>
<td>267</td>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
<td>dcprim</td>
<td></td>
</tr>
<tr>
<td>1261</td>
<td>1</td>
<td>readfname</td>
<td></td>
</tr>
</tbody>
</table>
Evaluation Completed

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

Statistics
System time was 4450 milliseconds
User time was 292650 milliseconds

----------------------------------
<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcprim</td>
<td>34</td>
</tr>
<tr>
<td>cons</td>
<td>10305</td>
</tr>
<tr>
<td>readiden</td>
<td>121</td>
</tr>
<tr>
<td>readint</td>
<td>3</td>
</tr>
<tr>
<td>letrec</td>
<td>18</td>
</tr>
<tr>
<td>eval</td>
<td>457</td>
</tr>
<tr>
<td>GT</td>
<td>92</td>
</tr>
<tr>
<td>sum</td>
<td>91</td>
</tr>
<tr>
<td>null</td>
<td>547</td>
</tr>
<tr>
<td>sin</td>
<td>91</td>
</tr>
<tr>
<td>cos</td>
<td>91</td>
</tr>
<tr>
<td>tan</td>
<td>91</td>
</tr>
</tbody>
</table>

150
Results of trig table generator (Compiled)
Evaluation Completed

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

Statistics
System time was 1216 milliseconds
User time was 24100 milliseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Cells created</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldcprim</td>
<td>84</td>
</tr>
<tr>
<td>cons</td>
<td>10305</td>
</tr>
<tr>
<td>readiden</td>
<td>121</td>
</tr>
<tr>
<td>readint</td>
<td>3</td>
</tr>
<tr>
<td>letrec</td>
<td>18</td>
</tr>
<tr>
<td>GT</td>
<td>92</td>
</tr>
<tr>
<td>sum</td>
<td>91</td>
</tr>
<tr>
<td>eval</td>
<td>457</td>
</tr>
<tr>
<td>null</td>
<td>547</td>
</tr>
<tr>
<td>tanr</td>
<td>91</td>
</tr>
<tr>
<td>cosp</td>
<td>91</td>
</tr>
<tr>
<td>sing</td>
<td>91</td>
</tr>
</tbody>
</table>

Profile for trigtable generating function

Berkeley Pascal PXP -- Version 2.12 (5/11/83)
Profiled Thu Dec 13 09:36 1984

Line    Count

151
<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>456</td>
<td><code>printval</code></td>
</tr>
<tr>
<td>36</td>
<td>11935</td>
<td><code>cellcount</code></td>
</tr>
<tr>
<td>63</td>
<td>88505</td>
<td><code>nullp</code></td>
</tr>
<tr>
<td>84</td>
<td>189153</td>
<td><code>first</code></td>
</tr>
<tr>
<td>94</td>
<td>102572</td>
<td><code>rest</code></td>
</tr>
<tr>
<td>122</td>
<td>10305</td>
<td><code>cons</code></td>
</tr>
<tr>
<td>156</td>
<td>91</td>
<td><code>sum</code></td>
</tr>
<tr>
<td>330</td>
<td>92</td>
<td><code>GTp</code></td>
</tr>
<tr>
<td>340</td>
<td>92</td>
<td><code>GT</code></td>
</tr>
<tr>
<td>411</td>
<td>91</td>
<td><code>sinp</code></td>
</tr>
<tr>
<td>433</td>
<td>91</td>
<td><code>cosp</code></td>
</tr>
<tr>
<td>454</td>
<td>91</td>
<td><code>tann</code></td>
</tr>
<tr>
<td>539</td>
<td>10495</td>
<td><code>atomp</code></td>
</tr>
<tr>
<td>547</td>
<td>88505</td>
<td><code>nullp</code></td>
</tr>
<tr>
<td>564</td>
<td>547</td>
<td><code>null</code></td>
</tr>
<tr>
<td>599</td>
<td>5563</td>
<td><code>assoc</code></td>
</tr>
<tr>
<td>643</td>
<td>2284</td>
<td><code>pairlis</code></td>
</tr>
<tr>
<td>663</td>
<td>456</td>
<td><code>printval</code></td>
</tr>
<tr>
<td>702</td>
<td>211</td>
<td><code>readval</code></td>
</tr>
<tr>
<td>704</td>
<td>298</td>
<td><code>nonblank</code></td>
</tr>
<tr>
<td>714</td>
<td>87</td>
<td><code>readlist</code></td>
</tr>
<tr>
<td>770</td>
<td>3</td>
<td><code>readint</code></td>
</tr>
<tr>
<td>827</td>
<td>510</td>
<td><code>digit</code></td>
</tr>
<tr>
<td>835</td>
<td>634</td>
<td><code>letter</code></td>
</tr>
<tr>
<td>843</td>
<td>121</td>
<td><code>readident</code></td>
</tr>
<tr>
<td>867</td>
<td>211</td>
<td><code>readval</code></td>
</tr>
<tr>
<td>879</td>
<td>639</td>
<td><code>evcon</code></td>
</tr>
<tr>
<td>880</td>
<td>8118</td>
<td><code>evlis</code></td>
</tr>
<tr>
<td>881</td>
<td>3646</td>
<td><code>apply</code></td>
</tr>
<tr>
<td>882</td>
<td>3</td>
<td><code>letrec</code></td>
</tr>
<tr>
<td>884</td>
<td>10495</td>
<td><code>eval</code></td>
</tr>
<tr>
<td>947</td>
<td>3</td>
<td><code>letrec</code></td>
</tr>
<tr>
<td>1012</td>
<td>639</td>
<td><code>evcon</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1041</td>
<td>8118</td>
<td>evlis</td>
</tr>
<tr>
<td>1093</td>
<td>2550</td>
<td>applyprim</td>
</tr>
<tr>
<td>1198</td>
<td>3646</td>
<td>apply</td>
</tr>
<tr>
<td>1239</td>
<td>28</td>
<td>dcprim</td>
</tr>
<tr>
<td>1261</td>
<td>1</td>
<td>readfname</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Program</th>
<th>MM</th>
<th>No MM</th>
<th>Left</th>
<th>System</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TIME</td>
<td>TIME</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reverse</td>
<td>381</td>
<td>401</td>
<td>46</td>
<td>416</td>
<td>9566</td>
</tr>
<tr>
<td>Revaux</td>
<td>427</td>
<td>427</td>
<td>46</td>
<td>533</td>
<td>1090</td>
</tr>
<tr>
<td>Append</td>
<td>531</td>
<td>561</td>
<td>51</td>
<td>566</td>
<td>10583</td>
</tr>
<tr>
<td>Map(sine)</td>
<td>599</td>
<td>629</td>
<td>95</td>
<td>533</td>
<td>12233</td>
</tr>
<tr>
<td>Halving</td>
<td>774</td>
<td>898</td>
<td>102</td>
<td>816</td>
<td>15800</td>
</tr>
<tr>
<td>Collate</td>
<td>718</td>
<td>748</td>
<td>72</td>
<td>700</td>
<td>17566</td>
</tr>
<tr>
<td>Factorial</td>
<td>427</td>
<td>457</td>
<td>18</td>
<td>416</td>
<td>7566</td>
</tr>
<tr>
<td>Interval</td>
<td>1042</td>
<td>1192</td>
<td>211</td>
<td>933</td>
<td>34483</td>
</tr>
<tr>
<td>Filter</td>
<td>755</td>
<td>776</td>
<td>147</td>
<td>516</td>
<td>14233</td>
</tr>
<tr>
<td>Periodic</td>
<td>1071</td>
<td>1393</td>
<td>8</td>
<td>1116</td>
<td>53416</td>
</tr>
<tr>
<td>Split</td>
<td>611</td>
<td>670</td>
<td>24</td>
<td>466</td>
<td>12300</td>
</tr>
</tbody>
</table>
(*Extended Lambda Calculus Interpreter*)

```c
program func(input, output);

(type

filename = packed array (.1..80.) of char;
tagtype = (lst, int, real, alf, bool);
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);
    real: (rval: real);
    alf: (aval: alpha);
    bool: (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, simp, *
*      cosp, tann, cot, sec, csc, atom, len, readea, 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, simp, *
*      cosp, tann, cot, sec, csc, atom, null, len, readea, *
*      readint, readstring, readident, eval, letrec, memb
*
function freecell: list;
  var C: list;
  begin
    C := h dr. next;
    h dr. next := C.d. tail;
    h dr. numcells := h dr. 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, apply, prim, 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);
        decr(C);
        return(C);
      end
      else if (C@.tag = alf) | (C@.tag = int) |
(C3.tag = rea) | (C3.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, cont, assoc, readlist, eval, letrec,
*            applyprim, apply
******************************************************************************)

procedure pptrassn (var x: list; y: list);
begin
  if diags then begin
    writeln('ref cnts of x and y in pptrassn if thy arent nil');
    if x <> nil then
      writeln(' x --> ', x3.ref);
    if y <> nil then
      writeln(' y --> ', y3.ref);
  end
end
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: cour, 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, dcrem *
*******************************************************************************

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 \& .c begin
    ref := 0;
    tag := int;
    rval := 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 \& .c begin
    ref := 0;
    tag := rea;
    rval := a\&.rval * b\&.rval;
  end;
  prod := P;
end
end;
subt := s;
end
else
  errormsg('subt', 3) (*Type mismatch*)
end; (*Function subtype*)

(*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:a.tag = int) and (b:a.tag = int) then begin
      if empty then begin
        new(P, int);
        cellcount(1, 'prod');
      end;
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 := ref;
a. 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 always subtracts the second argument from the first.

* Calls: empty, cellcount, freecell, errormsg

* Called by: applyprim

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

function suit (a,b: list): list;
var s: list;
begin
function sum(a, b: list): list;
  var k, l: list;
  begin
    if (a.tag = int) and (b.tag = int) then begin
      if empty then begin
        new(l, 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 = real) and (b.tag = real) then begin
      if empty then begin
        new(k, real);
        cellcount(1, 'sum');
        end
      else
head := T;
writeln('In conr 21');
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 result is returned through a pointer to another cell, created in the function. *
* Calls: celicount, freecell, errormsg *
* Called by: applyprim *)
function conf(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
        conf := cons(T, nil)
    else begin
        writeln('In conf 1st call');
        ptrasn(P, H);
        while P@.tail <> nil do
            P := P@.tail;
        if empty then begin
            new(C, lst);
            cellcount(1, 'conf');
        end
        else
            C := freecell;
        with C@ do begin
            ref := 0;
            tag := lst;
            end
        end
    end
end;
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 elements, 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(1);
    decr(0);
    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(M,T: list): list;
    var C: list;
    begin
        if empty then begin
            new( C, lst);
            cellcount(1, 'cons');
            end
        else
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);
      pptrassn(P, P@.tail);  
    until P@.tail = nil;
    O@.tail := nil;
    initial := L
  end
  else
    errormsg('initial', 2);
end

* Purpose: Receives a list as an argument and returns all elements of that list except the last element.

* Calls: pptrassn, decr, errmsg

* Called by: applyprim

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

*
end; (*Function first*)

FUNCTION rest

Purpose: Receives a list as an argument and returns all elements of that list except the first element.

Calls: errmsg

Called by: sub, lenp, assic, pairlis, printval, eval, evcon, membp, applyprim, apply

function rest (L: list): list;
begin
  if L = nil then
    errmsg('rest', 1)
  else if L@.tag = lst then
    rest := L@.tail
  else
    errmsg('rest', 2);
end; (*Function rest*)

FUNCTION initial
last := L@.head
end
else
   errorsj('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
      errorsj('first', 1)
   else if L@.tag = lst then
      first := L@.head
   else
      errorsj('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 errmsg*)

(*---------------------------------------------------------------------*)
* 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
  errmsg('last', 1)
else if L@.tag = lst then begin
  repeat
    ptrassn(L, L@.tail);
  until L@.tail = nil;
end;
* Function errorsmsg

* 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, sub, prod, divi, sub, LT, GT, GE, LE, simp, *
* COSP, TANN, COT, SEC, CSC, ASSOC, EVAL EVCON, ISFINSET, *
* applyprim

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

procedure errorsmsg(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 "real" 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
  errorsj('prod', 3) (*Type clash*)
end; (*Function prod*)

(*------------------------------------------------------------------*)
* Function divi
*
* Purpose: Divides two numbers
*
* Calls: empty, cellcount, freecell, errorsj
*
* Called by: applyrim
(*------------------------------------------------------------------*)

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;
      end
    else
      if (a.tag = int) and (b.tag = cell) then begin
        new(D, int);
tař := int;
ival := ař.ival div bř.ival;
end;
divi := D;
end (*if (ař...*)
else if (ař.tař = rea) and (bř.tag = rea) then begin
if empty then begin
new(D, rea);
cellcount(1, 'divi');
end
else
D := freecell;
with Dø do begin
ref := 0;
tag := rea;
rval := ař.rval / bř.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: errmsg, 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  
      errmsg('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;  
  end;
sub := first(L);
writeln('Finally 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 = real) and (y.tag = real) 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
    x0.ref := x0.ref + 1;
    y0.ref := y0.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;
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 = real) and (y.tag = real) then
    LTP := x.ival < y.ival
  end; (*Function LTP*)

(*---------------------------------------------------------------------*)
(* Function LT (Less Than)                                             *)
(* Purpose: Creates a cell that hold the Boolean response of calls to  *)
(*          LTP.                                                      *)
(* Calls: empty, cellcount, freecell, errmsg, 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 = real) and (y.tag = real)) then begin
      if empty then begin
        new(A, too);
        cellcount(1, 'LT');
      end;
      LTP := x.ival < y.ival;
    end; (*Function LTP*)
end
else
  A := freecell;
with A do begin
  ref := 0;
tag := hoo;
bval := LTr(x,y)
end;
LT := A
end
else
  errmsg('LT', 3);
decr(x);
decr(y);
end; (*Function LT*)

(**********************************************************************
* Function GTp (Greater Than)                                           *
* *
* Purpose: Determines if argument one is greater than argument two.     *
* *
* Calls: errmsg
* *
* 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 = real) and (y.tag = real) then
    GTp := x.ival > y.ival
end: (*Function GTp*)

******************************************************************************
* Function GT (Greater Than) *
* *
* Purpose: Creates a cell for the Boolean responses of calls to function 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)) or
     ((x.tag = real) and (y.tag = real)) then begin
    if empty then begin
      new( A, bool);
    end; (*begin*)
  end; (*if*)
end: (*function GT*)
cellcount(1, 'GT');
end
else
A := freecell;
with A do begin
ref := 0;
tag := 100;
val := 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 = real) and (y.tag = real) then
        GEp := x.rval >= y.rval
end; (*Function GEp*)

(*-----------------------------------------------------------*
* Function GE                                               *
*                                                          *
* Purpose: Creates a cell to hold Boolean responses from call to function GEp.
*                                                          *
* Calls: empty, cellcount, freecell, errors;j               *
*                                                          *
* Called by: applyprim                                       *
*-----------------------------------------------------------*)

function GE(x, y:list): list;
var A:list;
begin
    if ((x.tag = int) and (y.tag = int))
      or ((x.tag = real) and (y.tag = real)) then begin
        if empty then begin
            new(A, 0);
ceilcount(1, 'GE');
end
else
  A := freecell;
  with A do begin
    ref := 0;
    tag := too;
    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 argument 2
* *
* Calls: errormsg
* *
* Called by:  
*******************************************************************************)
function LEP(x,y: list): boolean;
begin
  if (x.thead = int) and (y.thead = int) then
    LEP := x.ival <= y.ival
  else if (x.thead = real) and (y.thead = real) then
    LEP := x.realval <= y.realval
end; (**Function LEP*)

(* Function LE *)**

* Purpose: Creates a cell to hold Boolean responses from calls to function LEP.

* Calls: empty, ccellcount, freecell, errorsj

* Called by: applyprim

function LE(x,y: list): list;
begin
  var A: list;
  if ((x.thead = int) and (y.thead = int))
  or ((x.thead = real) and (y.thead = real)) then begin
    if empty then begin
      new(A, 100);
    end;
  end;
end;
cellcount(1, 'LEX');
end
else
A := freecell;
with A do begin
ref := 0;
tag := 00;
lval := LEP(x,y)
end;
LEX := A
end
else
errormsg('IE', 3)
end; (*Function IE*)

(* Function sinp *)

* Purpose: Determines the sin of an angle in degrees.

* Calls: empty, cellcount, freecell, errormsg

* Called by: applytrim

function sinp(x:list):list;
var A: list;
    rad: real;
begin
    if (x.tag = int) or (x.tag = real) then begin
        (* Convert degrees to radians to use built-in trig functions *)
        case x.tag of
            int: rad := (xival * 3.141592654) / 180.0;
            real: rad := (x.rval * 3.141592654) / 180.0;
        end; (*case x.tag*)
        if empty then begin
            new(A, rad);
            cellcount(1, 'sinf');
        end
        else
            A := freecell;
        with A do begin
            ref := 0;
            tag := real;
            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 = real) then begin
      case x.tag of
        int: rad := (x.ival * 3.141592654)/180.0;
        real: rad := (x.rval * 3.141592654)/180.0;
      end; (*case x.tag*)
      if empty then begin
        new(A, real);
        cellcount(1, 'cos');
      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, errmsg
*
* 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@.real * 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)/ccs(rad);
end;
  tann := A;
end (*if (x@.tag...*)
else
  error$('tann', 5) (*Can only take the tan of a real or integer*)
end;  (*Function tanr*)

(***********************************************************************************

* Function cot (cotangent)  

***********************************************************************************)
function cot(x:list):list;
var A:list;
   rad:real;
begin
   if (x\a.tag = int) or (x\a.tag = real) then begin
      case x\a.tag of
         int: rad := (x\a.i\val * 3.141592654)/180.0;
         real: rad := (x\a.r\val * 3.141592654)/180.0;
      end; (*case x\a.tag*)
      if empty then begin
         new(A, real);
         cellcount(I, 'cot');
      end
   else
      A := freecell;
   with A\a do begin
      ref := 0;
   end
end
tag := rea;
  rval := 1/(\sin\(\text{rad})/\cos\(\text{rad})\);
end;

cot := A;
end (*if (\text{x@.tag}...*)
else
  errmsg('cot', 5) (*Can only take the \text{cot} of a real or integer*)
end; (*Function \text{cot}*)

([^*]********************************************************************
  * Function sec (Secant)
  *
  * Purpose: Determines the secant of an angle given in degrees.
  *
  * Calls: empty, cellcount, freecell, errmsg
  *
  * Called by: applyprim

  *********************************************************************)
function sec(x:list):list;
var A:list;
  rad:real;
begin
  if (\text{x@.tag = int}) or (\text{x@.tag = rea}) then begin
    case \text{x@.tag} of
      int: rad := (\text{x@.ival} * 3.141592654)/180.0;
    end
  end
end;
rea: rad := (xø.rval * 3.141592654)/180.0;
end; (*case xø.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 (xø.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@.real * 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
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, len, 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 atom.
*
* Calls: empty, cellcount, freecell, atomp
*
* Called by: applyprim
---------------------------------------------------------------------*)

function atom (L:list): list;
    var B: list;
    begin

if empty then begin
    new (B, lwo);
    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 interact 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;
rvkl := 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 interact then
        read(ch)
    else
        read(infile,ch);
    while (ch >= '0') and (ch <= '9') do begin
        n:= n*10 + ord(ch) - ord('0');
        k:= i * (1/(exp(expo * ln(10))));
        expo := expo +1;
        if interact 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 (Readrea) *)
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; (*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 '1st' 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
        1st: begin
          L := ls;
          write('<');
          if diags then begin
            write(' /tag is 1st, 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*
\texttt{decr(t);}  
\texttt{end; (*Function assoc*)}  

(*---------------------------------------------------------------------*)  
\texttt{* Function pairlis}  
\texttt{*}  
\texttt{* Purpose: Given a list of bound variable, \texttt{v}, and a list of actual}  
\texttt{* values, \texttt{x}, \texttt{pairlis} binds the variables with the actuals by}  
\texttt{* creating lists of pairs (attribute value pairs). These}  
\texttt{* lists of pairs are then added to the current environment, \texttt{A},}  
\texttt{* for use in the evaluation of a function call.}  
\texttt{*}  
\texttt{* Calls: nullp, cons, first, decr, rest}  
\texttt{*}  
\texttt{* Called by: eval, apply}  
(*---------------------------------------------------------------------*)  

\texttt{function pairlis \texttt{(v, x, A: list):list;}}  
\texttt{begin}  
\texttt{  if \texttt{v} <> \texttt{nil} then}  
\texttt{    v\texttt{.ref} := v\texttt{.ref} + 1;}  
\texttt{  if \texttt{x} <> \texttt{nil} then}  
\texttt{    x\texttt{.ref} := x\texttt{.ref} + 1;}  
\texttt{  if \texttt{A} <> \texttt{nil} then}  
\texttt{    A\texttt{.ref} := A\texttt{.ref} + 1;}  
\texttt{  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;
decl(A);
decl(x);
decl(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
    error msg('assoc', 1)
else if x.tag<>alf then
    error msg('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);
        end
        (*len(L)*);
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 interact then
read(ch)
else
  read(infile,ch);
i := 1;
a := ' ';
while ch <> '"""" do begin
  a(i,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;
  a val := a
end;
if interact 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 combination), 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 interact then
                read(ch)
            else
                read(infile, ch);
        end; (*while ch*)
        if empty then begin
            new(A, alf);
            cellcount(1, 'readident');
        end
        else
            A := freecell;
        with A do begin
            ref := 0;
        end
    end;
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 recognizes the first letter of the program and determines if a list, integer, real, or identifier is being read.
* Calls: readlist, readstring, readident, readint
* Called by: readlist, main program
*************
function readval;
beg
    if ch = '<' then
        readval := readlist
    else if ch = '*' then
        readval := readstring
    else if letter(ch) then
        readval := readident
    else
        readval := ch
end;
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, *
*        errors;
* 
* 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 atom(e) then eval := e
else begin
 trassn(e1p, first(e));
e1 := e1p@.aval;
if e1 = 'list' then eval := evlis(rest(e), a)
else if e1 = 'finsit' 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 \( e_1 = \text{'lambda'} \) 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 \( e_1 = \text{'if'} \) then eval := evcon( rest(e), a)
else if \( e_1 = \text{'call'} \) then
  eval := apply( eval( first( rest(e) ) , a) , evals( rest(rest(e)) , a))
else if \( e_1 = \text{'apply'} \) then
  eval := apply( eval( first( rest(e) ) , a) , eval( first( rest(rest(e)) ) , a))
else if \( e_1 = \text{'let'} \) then begin
  (*if diags then begin
    writeln('rest(e)');
    printval(rest(e));
    writeln;
  end;
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(rest(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;
writeln;
end
end;
decr(e);
decr(a);
decr(e1p);
end (*Function eval*)

************ Function letrec ************

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);
it 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(E, lst); cellcount(1, 'letrec') end
else E := 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 recursive
call of the function*)
with C0 do begin
   ref := 0;
tag := alf;
daiv := '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 begin
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*)
ptransn(f1p, first(f));
f1 := f1p@.aval;
if f1 = 'prim' then
  apply := applyprim(rest(f), x)
else begin
  temp := eval( first(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;
dcr(f);
dcr(x);
(*dcr(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 environment before evaluation.

* Calls: mtrassn, 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;
  end;
end;
else if \( f_1 = '\text{GT}' \) then
    applyprim := GT(first(x), first(rest(x)))
else if \( f_1 = '\text{LT}' \) then
    applyprim := LT(first(x), first(rest(x)))
else if \( f_1 = '\text{GE}' \) then
    applyprim := GE(first(x), first(rest(x)))
else if \( f_1 = '\text{LE}' \) then
    applyprim := LE(first(x), first(rest(x)))
else if \( f_1 = '\text{sin}' \) then
    applyprim := sinp(first(x))
else if \( f_1 = '\text{cos}' \) then
    applyprim := cosp(first(x))
else if \( f_1 = '\text{tan}' \) then
    applyprim := tanp(first(x))
else if \( f_1 = '\text{cot}' \) then
    applyprim := cotp(first(x))
else if \( f_1 = '\text{sec}' \) then
    applyprim := secp(first(x))
else if \( f_1 = '\text{csc}' \) then
    applyprim := cscp(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;
writeln;
printval(first(rest(x)));
writeln;
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@.avail;
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@.awal = '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, subtr, prod, div, 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 ',f1@.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): bcolean;
  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
        (* Further code for isfinset function *)
  end
end;
* 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;
    war C:list;
    begin
        if L = nil then
            errmsg('memb', 1)
        else begin
            if empty then begin
                new(C, 1);
                cellcount(1,'memb');
            end
            else
                C := freecell;
            with C do begin
                ref := 0;
                tag := 100;
                bval := membp(x, 1);
else
    evlis := cons(eval(first(L), a), evlis(rest(L), a));
end; (*Function evlis*)

****************************************************************************************************
*  Function memb
  *
  * 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 memb(x, L:list):boolean;
begin
    if nullp(L) then
        memb := false
    else if equalp(x, first(L)) then
        memb := true
    else
        memb := memb(x, rest(L))
end; (*Function memb*)
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: ');
    println(B);
  end;
  if B=nil then
    errmsg('evcon', 8)
  else if B.tag <> boo then
    errmsg('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;
end
ptrassn(head, R);
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: Evaluates a conditional expression (consists of three              *
*         sublists: condition, true consequent, and false consequent) by first determining the result of the condition  *
*         and then evaluating the true consequent or the false consequent.    *
*                                                                             *
* Calls: eval, errormsg, first, rest, decr                                  *
*                                                                             *
with D\theta do begin
    ref := 0;
    tag := alf;
    aval := 'prim';
end;
with E\theta do begin
    ref := 0;
    tag := alf;
    aval := primname;
end;

dcprim := cons(cons(C, ccns(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;

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('mem', primitives);
primitives$.ref := 1;
writeln;
diags := false;
terac := 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 ', dtmoyr, ',', 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 := readname;
  reset(infile, filearg);
end
else if argc = 2 then begin
  argv(1, filearg);
  if filearg = 'i' then begin
    interact := true;
    writeln('Note: '!!' ends an interactive terminal session');
  end
else
  reset(infile, filearg)
end;
writeln('Total program trace? type in '!!');
writeln('Monitor function calls only? type in '!!');
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 instructions 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


BIBLIOGRAPHY


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


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


254
<table>
<thead>
<tr>
<th>No.</th>
<th>Distribution List</th>
<th>Copies</th>
</tr>
</thead>
</table>
| 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      |