The LispKit Manual
Volume 1

Peter Henderson
University of Stirling

Geraint A. Jones
Oxford University

Simon B. Jones
University of Stirling

[α] Department of Computing Science
University of Stirling
Stirling   FK9 4LA
Scotland

[β] Jesus College
Oxford   OX1 3DW
England

Introduction

LispKit is the portable implementation technique for a purely functional programming language, LispKit Lisp. The implementation consists of a small virtual machine, and a corpus of language support software, itself written in LispKit Lisp. It is the size and simplicity of the virtual machine which gives the implementation its portability.

The language is a dialect of the Lisp language and shares two of the most powerful features of that language: the universal structured type, the list; and s-expression syntax which is both expressive and easily manipulated by program. It differs from full Lisp in that all LispKit programs must be written in a functional style, since there are no language primitives for updating data structures. The absence of destructive assignment greatly improves the intelligibility of programs, and is held to improve programmer productivity, by simplifying the creation and maintenance of correct programs.

The principal text both on the LispKit implementation, and on the LispKit Lisp dialect, and a useful introduction to this manual, is

Functional Programming,   Application and Implementation,   P. Henderson
Prentice/Hall International, London 1980, 0-13-331579-7

referred to throughout this manual as *the book*. It describes a virtual machine and a language slightly different from those described in this manual, but familiarity with the system described in the book will help in an understanding of the current system, especially so if it is intended to make any changes to the virtual machine or to the system support software.

In order to install the LispKit system on your machine, first follow the instructions in the section at the back of this manual which is specific to your particular machine. That section also contains a detailed description of machine specific features such as the interface to the filing system, and the precision of arithmetic. Having followed these instructions read the first part of the manual, which takes you through the writing of a simple program, its compilation and execution.

The substance of the manual follows, consisting of a detailed description of the construction, function, and use of each of the components of the support software. The companion volume to this manual contains source listings of each of the programs described in the manual, and may be useful for reference.

Finally, the implementor's guide describes those features of the virtual machine which differ from that described in the book, together with details of the scheme used to manipulate compiled code objects in the LispKit system. Reference to this section should be unnecessary unless you intend to make changes to the implementation, and you should be warned that the details in this section may be changed in future issues of the system.

<div align="right">

G. A. J.

</div>

Summary of the LispKit Lisp language

Every LispKit Lisp program is an expression — an s-expression — which has a value, and this value depends only on the values of the components of the expression, which are themselves LispKit Lisp expressions. The syntax of the language is simple: every expression is either the name of a variable, or is one of the forms of expression beginning with a keyword as described below, or is a list of components, each of which is an expression.

The semantics of the language are also simple: the meaning of any expression is its value, no more and no less, and this value depends only on the values of its component expressions; any component sub-expression of an expression may be replaced by any other expression having the same value without changing the value of the whole. Generally speaking, sub-expressions are evaluated only if their values become necessary in the evaluation of a larger expression. This means that infinite values — for example the list of all the numbers, or of all the digits in the decimal expansion of an irrational number — may be handled in a natural way.

Data types in LispKit Lisp

There are four types of value which a LispKit Lisp expression may have. The simplest types of expressions have values which are either numbers, or symbols.

In the case of the system to which this manual applies, a number is a signed integer in a machine specific finite range, and the representation of a number is a sequence of decimal digits, optionally preceded by a sign. For example,

    –0   0   and   +0

all represent the number zero, and each of

    45   +137   and   –27

is a distinct number.

A symbol is represented by a finite sequence of characters. Since the system interprets any number representation in its input as a number, it is not possible for the user to type symbols which begin as though they were numbers, although digits may appear later in a symbol that begins with, say, a letter. For reasons that will become apparent, neither can the user type symbols which contain spaces, newlines, full stops, or opening or closing parentheses. Examples of symbols are

    Hello   hello   Hello_world   X32   X+32   :**@?

each of which is distinct from the others.

Both numbers and symbols are called atomic values, since they have no internal structure accessible to the LispKit programmer. Essentially, atoms are values which may be compared for equality; in addition, the usual arithmetic operations are provided for the manipulation of numbers.

There is a single primitive data structure in LispKit Lisp, namely the pair. A value which is a pair is distinct from any atomic value, and consists of two component values, called the head and the tail of the pair. The components of a pair may be any type of value, including being themselves pairs, so that pairing may be used to construct arbitrarily large data structures. One particular form of data structure is known as a list: there is one atomic list, the symbol NIL; any pair is a list provided that its tail is also a list. Thus a list is either NIL, or contains a sequence of components which are

　* its head
　* the head of its tail
　* the head of the tail of its tail

and so on, terminated by a tail which is NIL. We shall normally use the term 'list' only for finite sequences of this form, and the term 'stream' for sequences with infinitely many components. The representation of a pair consists of the representations of the components, separated by a full stop, and the whole enclosed in parentheses, thus for example, each of

　(a.b)　(−123.(p.q)) and　(((a.b).c).(d.e))

is a pair; the first is a pair of symbols, the others are pairs which have pairs as components.

There is an abbreviated representation which is convenient in the case of lists: it is permitted to omit the full stop and the parentheses around the tail of a list, or the full stop and the NIL in the tail of a list, so that each of

　(a.(b.(c.(d.()))))
　(a.(b.(c.(d.NIL))))
　(a.(b.(c.(d))))
　(a.(b.(c d)))
　(a.(b c d))
　(a b c d)

represents the same list. Layout — spaces and newlines — may appear anywhere between atoms in input to the LispKit system, and is ignored, except that atoms must be separated from each other, either by punctuation — a full stop or an opening or closing parenthesis — or by at least one space or newline.

The fourth form of data value is the function. There is no representation for a function value, but there are LispKit Lisp expressions whose values are functions, as explained below. A function is a value which may validly be applied to arguments, to yield a value which depends only on the function and the arguments to which it is applied.

Expressions in LispKit Lisp

The simplest expression in LispKit Lisp is the constant expression, which has the form

```
(quote ⟨exp⟩)
```

Its value is the expression ⟨exp⟩, so that the values of the expressions

```
(quote 0) (quote 1) (quote 2) (quote 3)
```

are the first four natural numbers, and the value of the expression

```
(quote (Hello world!))
```

is a list with two components, the symbol Hello and the symbol world!.

The equality of two atomic expressions may be tested by an expression of the form

```
(eq ⟨exp1⟩ ⟨exp2⟩)
```

whose value is the symbol T, provided that the values of ⟨exp1⟩ and of ⟨exp2⟩ are both atoms, and either they are both the same number, or they are both the same symbol; its value is the symbol F otherwise. In particular, note that if either of the expressions has a value which is not an atom, the value of the eq-expression must be F. The expression

```
(atom ⟨exp⟩)
```

has value T if the value of ⟨exp⟩ is an atom — a number or symbol — and has value F otherwise.

Expressions which take the values T and F — standing for True and False — are principally used to control the choice between alternative values. The value of an expression of the form

```
(if ⟨exp1⟩ ⟨exp2⟩ ⟨exp3⟩)
```

is the value of the expression ⟨exp2⟩ provided that the value of ⟨exp1⟩ is the symbol T, and is the value of ⟨exp3⟩ otherwise. For example,

```
(if (atom (quote NIL)) (quote gwir) (quote anwir))
```

has value the symbol gwir, and

```
(if (eq (quote (a)) (quote (a))) (quote gwir) (quote anwir))
```

has value the symbol anwir, since the eq-expression has value F, neither of its components being an atom.

Pairs are constructed by expressions of the form

```
(cons ⟨exp1⟩ ⟨exp2⟩)
```

which is an expression whose value is a pair, the head of which is the value of ⟨exp1⟩, and the tail of which is the value of ⟨exp2⟩. Pairs may be analysed by expressions of the form

```
(head ⟨exp⟩)    and    (tail ⟨exp⟩)
```

whose values are the head and tail, respectively, of the value of ⟨exp⟩. If the value of ⟨exp⟩ is not a pair, then neither of these expressions is defined. Examples of these expressions are

```
(cons (eq (quote NIL) (quote NIL)) (quote (a.b)))
```

the value of which is (T.(a.b)), and

```
(head (cons (tail (quote (a b c))) (quote NIL)))
```

the value of which is (b c).

Large LispKit Lisp expressions may be made more easily readable by giving names to their components. This may also make their evaluation more efficient, since a frequently occurring expression need only be evaluated once if it is given a name that is subsequently used to stand for its value. The simplest way of naming components of an expression is the form

```
(let ⟨exp⟩ . ⟨declarations⟩)
```

where the ⟨declarations⟩ are a list of pairs

```
(⟨name⟩.⟨exp⟩)
```

The value of the expression

```
(let ⟨exp⟩ (⟨name1⟩.⟨exp1⟩) (⟨name2⟩.⟨exp2⟩) ···)
```

(which is read as 'let ⟨name1⟩ be ⟨exp1⟩ and ⟨name2⟩ be ⟨exp2⟩ ··· in ⟨exp⟩') is the value of its body — ⟨exp⟩ — where occurrences of the names ⟨name1⟩, ⟨name2⟩, ··· stand for the values of the corresponding right hand side expressions, ⟨exp1⟩, ⟨exp2⟩, ···. There may be any number of declaration pairs in a let-expression, for example, the value of each of

```
(let (quote (aleph (aleph beth) (aleph beth))))
```

and

```
(let (cons (quote aleph) (cons aleph (cons aleph beth)))
      (aleph.(quote (aleph beth)))
      (beth.(quote NIL)) )
```

is the list

```
(aleph (aleph beth) (aleph beth))
```

Note that the occurrence of the atom aleph in the expression

```
(quote aleph)
```

is not a name, but a constant, much like a quoted string in a Pascal program, and so is not relevant to the declarations in the let-expression.

Another way of naming sub-expressions is the function-valued expression

```
(lambda ⟨argument list⟩ ⟨exp⟩)
```

where the ⟨argument list⟩ is a list of names. The value of the expression

```
(lambda (⟨name1⟩ ⟨name2⟩ ···) ⟨exp⟩)
```

(which is read 'that function of ⟨name1⟩, ⟨name2⟩ ··· which is ⟨exp⟩') is a function. It is a function which may validly be applied to as many arguments as there are names in the ⟨argument list⟩. The value of such an application is the value of ⟨exp⟩, where occurrences of the names ⟨name1⟩, ⟨name2⟩, ··· stand for the values of the corresponding arguments in the argument list.

A function application is an expression of the form

```
(⟨exp⟩ ⟨exp1⟩ ⟨exp2⟩ ···)
```

where the value of ⟨exp⟩ is the function. The value of the whole of such an application is the value of the body of the function where the values of the actual arguments, ⟨exp1⟩, ⟨exp2⟩, ··· are given to the formal arguments. The value of an application is not defined either in case the first expression, ⟨exp⟩, does not have a function value, or in case its value is a function of a number of arguments different from the number of actual arguments — ⟨exp1⟩, ⟨exp2⟩, ··· — provided.

Similar to let-expressions are letrec-expressions, which take the form

```
(letrec ⟨exp⟩ . ⟨declarations⟩)
```

The value of such an expression is the value of the body ⟨exp⟩, where the names introduced in the declarations stand for the values of the corresponding expressions. A letrec-expression differs from a let-expression only in that occurrences of the declared names in the right hand sides of the declarations also stand for the corresponding values. This means that letrec-expressions (letrec standing for 'let, recursively') may have values which are infinite, or which are recursively defined functions, for example, the value of

```
(letrec i (i.(cons (quote 1) i)))
```

is the stream consisting of an infinite sequence of ones:

```
(1 1 1 1 1 1 1 1 1 1 1 1 1 1 ···
```

and the value of

```
(letrec reverse
    (reverse.(lambda (l)
        (if (eq l (quote NIL)) (quote NIL)
                (append (reverse (tail l))
                        (cons (head l) (quote NIL))) ) ))
    (append.(lambda (a b)
        (if (eq a (quote NIL)) b
                (cons (head a) (append (tail a) b)) ) )) )
```

is the function which, given a list as its argument, reverses that list.

Several forms of expression are provided for performing arithmetic operations on numbers. Provided that the values $v1$ and $v2$ of ⟨exp1⟩ and ⟨exp2⟩ are numbers, and that the results of the operations are expressible

in the number range of the particular LispKit implementation, the value of

   (add ⟨exp1⟩ ⟨exp2⟩)

is the sum of v1 and v2; the value of

   (sub ⟨exp1⟩ ⟨exp2⟩)

is the difference v1 −v2; the value of

   (mul ⟨exp1⟩ ⟨exp2⟩)

is the product of v1 and v2; the value of

   (div ⟨exp1⟩ ⟨exp2⟩)

is the quotient obtained on dividing v1 by v2; the value of

   (rem ⟨exp1⟩ ⟨exp2⟩)

is the remainder on dividing v1 by v2; the value of

   (leq ⟨exp1⟩ ⟨exp2⟩)

is the symbol T if v1 is less than or equal to v2 and is the symbol F otherwise. The division operation is an integer division, and in the case of v1 and v2 both being positive, the value of

   (div ⟨exp1⟩ ⟨exp2⟩)

is less than or equal to their exact quotient, that of

   (rem ⟨exp1⟩ ⟨exp2⟩)

is less than v2, and that of

   (add (mul (div ⟨exp1⟩ ⟨exp2⟩) ⟨exp2⟩) (rem ⟨exp1⟩ ⟨exp2⟩))

is v1.

   Finally, if the value of ⟨exp⟩ is a number in the appropriate range, the value of the expression

   (chr ⟨exp⟩)

is the single character symbol consisting of the character with that ASCII code, except that chr of thirteen (ASCII carriage return) is translated by the LispKit system into the representation of a new-line, appropriate to the underlying machine, and the particular output device being used.

How to write and run a program

It is assumed that either you will have some experience of writing programs in a functional style, or at least you have available a text on the subject, such as the book. This manual does not attempt to perform the function of a text book: this chapter is intended only to introduce you to the keystrokes necessary to write and to run a LispKit Lisp program. Doing this will require you to use a number of the utility programs supplied as parts of the system, themselves LispKit Lisp programs. Again, this chapter does not detail the full capabilities of these programs, for example the editor, and you should certainly consult the particular descriptions of these utilities once you have succeeded in writing your first program.

The program which we will take as an example is one which reads a sequence of numbers from the keyboard. Whenever the sequence is interrupted by the atom *sum*, the accumulated sum of the numbers that have been input is to be output, and if the atom *end* is typed, the program is to terminate.

Designing the program

Every LispKit Lisp program is an expression whose value is a function. This function maps the potentially infinite sequence of input s-expressions onto the list of s-expressions which appear at the output. Thus, in our case, the function must map an input stream beginning

(2 2 sum 5 sum end ···

into the output list

(4 9)

It is probably good practice for a program to announce itself, and possibly to explain its output, so we will actually cause it to generate the output list

```
( Example program ⟨newline⟩
  Sum is 4 ⟨ newline⟩
  Sum is 9 ⟨newline⟩
  Finished )
```

The kernel of this program is therefore

```
(lambda (input_stream)
            (append (quote (Example program))
            (cons newline
            (append (add_up (until_end input_stream))
                    (quote (Finished)) )))) )
```

where we have yet to define the functions add_up and until_end. The function append and the value of newline will be obtained from a library of standard definitions.

Notice that the input stream is thought of as being infinite: the program will only read a finite number of s-expressions from the beginning of this stream. Notice also that although we have written parentheses around the input and output sequences to emphasise that we think of them as single entities, you will not type the initial parenthesis in the input, nor will the parentheses appear in the output from your program.

The function until_end takes a stream as an argument, and returns the list of those components of the stream which precede the first component which is the atom end. Its definition is

```
until_end = (lambda (s)
                  (if (eq (head s) (quote end))
                    (quote NIL)
                    (cons (head s) (until_end (tail s))) ))
```

The definition of add_up is the next task, and we choose to keep a running total of the numbers input so far, so the definition must be something like

```
add_up = (lambda (numbers) (accumulate numbers (quote O)))
```

where accumulate is a function, still to be written, which adds the numbers from its first argument to the running total, which is its second argument, and returns a list of the values of this accumulator at the points where *sum* occurs in the first argument. The definition we choose is

```
accumulate =
      (lambda (l total)
            (if (eq l (quote NIL)) (quote NIL)
            (if (eq (head l) (quote sum))
                  (append (print total) (accumulate (tail l) total))
                  (accumulate (tail l) (add (head l) total)) )) )
```

The function print simply formats an output line

```
print = (lambda (n)
                  (append (quote (Sum is))
                          (cons n (cons newline (quote NIL))))) )
```

Finally, we will put all these components together in a single expression using the letrec-form (letrec because both until_end and accumulate are recursive functions)

```
(letrec
      (lambda (input_stream) ···)
      (until_end. (lambda ···))
      (add_up. (lambda ··))
      (accumulate. (lambda ···))
      (print. (lambda ·· )) )
```

Inputting the text of the program

It is possible to prepare LispKit Lisp sources with a conventional text editor, and then to use the LispKit system only to compile and execute the code. This approach is not to be recommended, however, since you would spend a great deal of your time in counting parentheses, and might easily miscount! You would also be disregarding the facility with which the LispKit editor allows whole expressions to be manipulated as single entities. We will use the structure editor to input the text of the program, deliberately introducing an error, so that we may later use the editor to correct that error.

First, begin execution of the SECD machine. Details of how to do this will be found in the section of machine-specific information. On the Perq, for example, you must log in, set the default pathname or the search list so as to have access to a directory or directories containing both the machine and the bootstrap file, and type

fSECD⟨return⟩

The machine will then announce itself and its version number, and prompt you for input.

The machine reads all of its input from a single input stream which is normally connected either to a filing system file, or to the keyboard of your terminal. Whenever the machine requires more input, it will prompt

Take input from where?

to which you should reply with the name of the next file which the machine should read. There is a special name — usually CONSOLE:, or an empty line, that is just ⟨*return*⟩ — which you can give to cause the machine to treat the keyboard as its next input file. By now, the machine has already prompted you for its first input file. The reply to this prompt should be the name of the file containing the code of the editor. On the Perq, this file is E.LOB, on other machines it may be different, check the file name conventions in the section on machine specifics. You would then type something like

E.LOB⟨return⟩

The code having been read, it is executed, and the editor announces itself and prompts you for another file name

Editor
Take input from where?

This time, the editor is expecting the file to be edited, and since the input will come from the console, you type

CONSOLE:⟨return⟩

or whatever is appropriate to your particular machine. Now you are expected to type the file to be edited. This file is the s-expression which is the

source of the program. You could try typing the whole of the program at this point, but it is easier to type large s-expressions in small portions. We will choose to type the outer level of the program, and fill in the detail later, so you type

```
(letrec body until_end add_up accumulate print)⟨return⟩
```

The editor will now reply

```
ready
```

which is a signal that it has read in the whole of the file to be edited: in this case a check that the parentheses balance! Whenever the editor is expecting a command, if you type

```
p⟨return⟩
```

the editor will print an outline of the expression currently being edited, suppressing deeply nested fine detail. Type this now, and the editor will output

```
( letrec body until_end add_up accumulate print )
```

— there is as yet no fine detail to be suppressed. To change the body component of this expression, we will use the exchange command, for which you should type

```
(e body
    (lambda (input_stream)
        (append (Example program)
    (cons newline
        (append (add_up (until_end input_stream))
                        (quote (Finished)) ))) ) )
```

This command replaces the component of the list that matches body by the expression

```
(lambda (input_stream) ···)
```

Here we have introduced the deliberate error — omitting to quote the list

```
(Example program)
```

Layout between any two atoms, or between atoms and punctuation is totally insignificant, and you may choose to lay out your input as you please. When you have typed the exchange command, try typing

```
p⟨return⟩
```

again. This time, the output should look something like

```
( letrec
  (lambda ( input_stream ) ( append . * ) )
  until_end
  add_up
  accumulate
  print )
```

The asterisk represents structure which is nested in more than three levels of parentheses. If you get no output, then it is probably because you forgot to close an opening parenthesis in the exchange command! The editor will still be trying to read the rest of that command, and you will have to type enough closing parentheses to complete the command. The other components can now be changed in turn, again using exchange commands.

```
(e until_end
     (until_end.
          (lambda (s)
               (if (eq (head s) (quote end))
                    (quote NIL)
                    (cons (head s) (until_end (tail s)))) ) ))

(e add_up
     (add_up.
          (lambda (numbers) (accumulate numbers (quote 0))) ))

(e accumulate
     (accumufate.
          (lambda (I total)
               (if (eq I (quote NIL))
                    (quote NIL)
               (if (eq (head I) (quote sum))
                    (append (print total)
                         (accumulate (tail I) total))
                    (accumulate (tail I) (add (head I) total))) )) ))

(e print
     (print.
          (lambda (n)
               (append (quote (Sum is))
                    (cons n (cons newline (quote NIL)))) ) ))
```

You should probably check your typing by giving print commands between successive exchanges. If you now type a final print command, the editor should output something like

```
( letrec
  ( lambda ( input_stream ) ( append . * ) )
  ( until_end lambda ( s ) ( if . * ) )
  ( add_up lambda ( numbers ) ( accumulate numbers * ) )
  ( accumulate lambda ( I total ) ( if . * ) )
  ( print lambda ( n ) (append . * ) ) )
```

Notice that although you typed the definitions in the letrec-expression as dotted pairs, the editor has output them with the minimum of punctuation. You can of course, should you wish, use this form for input. If you now type

(p all)⟨return⟩

then the editor will print the whole of the program, suppressing no detail.

Now, try typing

file⟨return⟩

and the editor will print the whole of the program again, but this time without formatting the output. This is the form in which programs are stored in files.

In exactly the way that the machine has a single input stream which is used to read a succession of files, so too it has a single output stream which can be redirected on demand. Whenever the machine's input stream is connected to your keyboard, and the machine is waiting for input, you can type a key — usually 'control and Y', that is the 'Y' key struck whilst holding down the control key — which will cause the machine to prompt you

Send output to where?

The reply is again a file name, and from this point on all output from LispKit Lisp programs will be sent to that file, until you redirect the output stream again. There is, of course, a name — usually CONSOLE: or the empty name, again — which names the screen of your terminal.

To write the program into a file, type

⟨ctrl Y⟩⟨return⟩

and the LispKit system will prompt you with

Send output to where?

Here you should type the name of the file to which you want to send the program text, we shall choose EXAMPLE.LSO, but you should probably follow the file name conventions of your machine if this is different from ours. Type the file name

EXAMPLE.LSO⟨return⟩

The editor is still expecting a command from you, but whatever command you type at this point, all output from the editor will be sent to the file EXAMPLE.LSO. Naturally what you should type here is

file⟨return⟩

but be careful, because even error messages will go to the output file, and not to the screen!

Having done this, you will want to redirect the output to the screen, which is done by typing

⟨ctrl Y⟩⟨return⟩
CONSOLE:⟨return⟩

Output from the editor will now come to your screen, and you can carry on editing the same expression should you wish. In this instance, we are finished with the editor, so you should type

end⟨return⟩

Checking and correcting syntax errors

The LispKit system is now expecting the code of a new program to appear in the input stream. In order to cause this to be read from a file, rather than from the keyboard, it will be necessary to type a key that indicates to your machine that its current file, which is the keyboard, is now exhausted. This key is usually 'control and Z', but you should again check the details for your particular machine. Type

    ⟨*ctrl Z*⟩

without, notice, a ⟨*return*⟩ to follow it. The system will prompt you, as before.

    Take input from where?

to which you should reply with the name of the file containing the code of the syntax checker

    SYNTAX.LOB⟨*return*⟩

The syntax checker will then announce itself, and demand its input

    Syntax check
    Take input from where?

to which you should reply with the name of the file containing the program source

    EXAMPLE.LSO⟨*return*⟩

There will now follow a list of detected defects in your program, which in our case would be something in the nature of

```
append used but not defined
      in the body of the program
Example used but not defined
      in the body of the program
program used but not defined
      in the body of the program
newline used but not defined
      in the body of the program
append used but not defined
      in the body of the program
append used but not defined
      in accumulate
append used but not defined
      in print
newline used but not defined
      in print
```

Now we know that append and newline were meant not yet to be defined, we will find these in a standard library later, but the occurrences of Example and program are definitely errors, so we will go back to the editor to change the offending expression.

By now, the system is already prompting for an input file, so type

    E.LOB⟨return⟩

and the editor will announce itself, and prompt

    Editor
    Take input from where?

Recall that at this point, the editor expects the file which is to be edited, so type

    EXAMPLE.LSO⟨return⟩

and the editor will re-read the text which it had earlier written, and prompt

    Take input from where?

Since the editor now requires a command, you should reply

    CONSOLE:⟨return⟩
    p⟨return⟩

this last being a print command, to check that we are editing the right file. You will see that the body of the program, in which the cheeker found an error, is the second component of the file, so type

    2 p⟨return⟩

The command 2 makes the second component of the current expression the new object of attention, so that this print command will output

    (lambda
      (input_stream)
      (append (Example program) (cons newline *)))

The simplest sequence of commands that can repair the error is probably

    3 2 (c 99 (quote 99))

which first makes the sub-expression

    (Example program)

the current expression, and then uses the change command to replace it by

    (quote (Example program))

Notice the use of 99 in the change command: a number in a pattern stands for 'any expression', and then in a replacement. the same number stands for 'the same expression again'. You can now move back up through the structure of the expression using the up command

    u

and printing the expression as you go, should you want to inspect the change and make sure that you have it right. Finally, you will want to write the newly changed expression back out to a file. Use the top command

> *top*

to select the whole of the file being edited as the current expression, before you write out the file. The file command and the various print commands output only the current expression, so it is good practice always to close an edit session

> ⟨*ctrl Y*⟩⟨*return*⟩
> Send output to where? ⟨*file name*⟩⟨*return*⟩
> top file⟨*return*⟩
> ⟨*ctrl Y*⟩⟨*return*⟩
> Send output to where? *CONSOLE:*⟨*return*⟩
> *end*
> Exit editor

using a top command to ensure that you do not forget to write out the upper levels of the structure of an expression.

If you wish, you can now run the syntax checker again, and you will find that only standard functions have yet to be defined. We will move on to fetch these from a library.

Compiling the program

The next input which the machine requires is another program to be executed. Type

> ⟨*ctrl Z*⟩*LIBMAN.LOB*

to run the library manager, which will announce itself, and prompt for an input file. The input expected at this point is the function that you have written, so type

> *EXAMPLE.LSO*

The library manager will scan through your function looking for undefined names, and will prompt you with a list of these names, and request more input

> newline append
> Take input from where?

The input expected at this point is a library which defines newline and append. In this case, the relevant library is

> *STANDARD.LIB*

from which the librarian will extract the necessary functions. The next output from the librarian is

> chr

which is a list of names used but not defined in the expression formed by

adding the definitions of newline and append to your program. This function is defined in the library SECD_CODE, so reply to

> Take input from where?

with the file name

> *SECD_CODE.LIB*

The librarian would go on to prompt you with the list of names still undefined, but in this case there will be no more undefined names, so the next prompt will be

> Type anything to print result
> Take input from where?

Now you will need a work file to keep the program constructed by the library manager, and the interaction from this point on should go as follows

> Take input from where? *CONSOLE:⟨return⟩*   to take input from the keyboard
> ⟨*ctrl Y*⟩⟨*return*⟩
> Send output to where? *WORKFILE.LSO⟨return⟩*   to send output to the workfile
> *file⟨return⟩*   to cause the output to be sent
> ⟨*ctrl Y*⟩⟨*return*⟩   to close the workfile

The next program that you are going to run is the compiler. This generates the code as its only output, taking the source as its input, so the interaction proceeds

> Send output to where? *EXAMPLE.LOB⟨return⟩*
> ⟨*ctrl Z*⟩
>
> Take input from where? *LC.LOB⟨return⟩*
>
> Take input from where? *WORKFILE.LSO⟨return⟩*
>
> Take input from where? *CONSOLE:⟨return⟩*
> ⟨*ctrl Y*⟩⟨*return*⟩
> Send output to where? *CONSOLE:⟨return⟩*

Running the program

The program which you have written and compiled is exactly the same kind of object as the utilities that you have used, so you should be able to follow the following interaction without further commentary.

> ⟨*ctrl Z*⟩
>
> Take input from where? *EXAMPLE.LOB⟨return⟩*
> Example program
>
> Take input from where? *CONSOLE:⟨return⟩*
> *2⟨return⟩*
> *2⟨return⟩*

```
sum⟨return⟩
Sum is 4
5 sum⟨return⟩
Sum is 9
1 sum 1 sum 1 sum⟨return⟩
Sum is 10
Sum is 11
Sum is 12
end ⟨return⟩
Finished
```

Try running the program again, check that you know how to send either parts or the whole of the output to a file, and can cause the program to take its input from a file. Read the documentation on the editor, which you will find later in this manual, in the section headed *the list structure editor*; experiment with using the editor to check that you understand the more powerful editing commands.

When you have finished your LispKit session, you can return to the host operating system of your machine by executing the program HALT.LOB.

LispKit Lisp support software

The core of the programming environment is provided by four programs, a structure editor E, the compiler LC for the LispKit Lisp language, a source code constructor LIBMAN which is used in the modularisation of Lisp programs, and a syntax checker SYNTAX which makes detailed checks for statically detectable errors in Lisp programs.

A LispKit Lisp program is normally created with the editor, which is principally intended for the manipulation of the s-expression syntax of Lisp programs. Since libraries are also s-expressions, the editor may also be used to create libraries — effectively separate modules — containing component functions of the program. The program is then assembled using the library manager, LIBMAN, which extracts from each library just those functions required by the program, and constructs a self-sufficient source. This is then translated by the compiler, LC, into a code object which can be executed by the virtual machine. Since the same s-expression syntax is used for code, even code objects may be manipulated and inspected by the use of the editor.

The LispKit Lisp compiler

The LispKit Lisp compiler translates a LispKit Lisp source expression into an object which, when executed as code by the SECD virtual machine, evaluates that expression. The source expression may be any closed LispKit expression — that is, one in which every name which occurs is also defined. The code is also an s-expression, with a structure that reflects the structure of the definitions in the source, but where the operators of LispKit Lisp have been translated into SECD machine opcodes.

When the code produced by LC is executed, the expression is evaluated in the lazy order (call by first demand: see the book for a discussion of lazy evaluation). This means that interactive programs will take their inputs when they are first needed, and that infinite values may be handled simply. The meaning of the code which is output by the compiler is discussed in the implementor's guide. All that concerns us here is that it is a data structure with a finite s-expression representation which may be sent to a file.

The output of the compiler is indeed normally sent to a file, from which it may be retrieved for subsequent execution. By convention filenames with extension

.LOB

are used for the codes of complete programs, which are functions from an input stream to an output stream; filenames with extension

.CLS

are used for all other codes.

The LispKit compiler makes no checks on the validity of its input, so outputs no error messages; neither does it output any form of prompts. The first input must be the source program, and the first and only output is the compiled code. Accordingly, the interaction required to compile the INTEGERS expression would be as follows:

Take input from where? *CONSOLE:*   to allow output redirection
⟨*ctrl Y*⟩⟨*return*⟩
Send output to where? *INTEGERS.CLS*   destination for compiler output
⟨*ctrl Z*⟩

Take input from where? *LC.LOB*   compiler is program to be
executed

Take input from where? *INTEGERS.LSO*   the Lisp source

Take input from where? *CONSOLE*   to allow output redirection
⟨*ctrl Y*⟩⟨*return*⟩   to close the output file
Send output to where? *CONSOLE:*

Notice particularly that the input to the compiler must be a closed expression. Many of the program sources in the LispKit system have free variables which are meant to be references to libraries; these texts must be passed through the library manager before being compiled. This means that the input to the compiler is often taken from an intermediate workfile that has been produced as output from the library manager.

Since the compiler makes no checks on the expression supplied as its argument, it is quite likely to crash if there are errors — such as undefined names — in the source text. The syntax checker, SYNTAX, which is a part of the LispKit system, will find any errors that could cause the compiler to fail.

Construction notes

LC.LSO requires the following libraries

    LISPKIT.LIB
    OP_CODE.LIB
    ASSOCIATION.LIB
    STANDARD.LIB

and the composite text compiles to produce the closure in **LC.CLS**. The value represented by the closure is a function mapping the source onto the code; the complete program in **LC.LOB** consists of the following sequence of items

the closure	LOADS.CLS
the closure	LC.CLS
an argument count	1

The Lispkit Lisp syntax checker

The syntax checker is a program which checks for most statically detectable errors in a LispKit Lisp source expression. In particular, it will find all unbound variables, and any errors of form. Any expression which is passed as correct by the syntax checker can be compiled by the LispKit Lisp compiler.

Each detected error results in a message in the output stream, which should be essentially self explanatory. The general form of an error message is

⟨type of error⟩
    ⟨position⟩

where the ⟨type of error⟩ is a message such as

append used but not defined

or

incorrect form of definition

and the ⟨position⟩ is a sequence

in alpha in beta in gamma

which means that the error is to be found in the definition of alpha which occurs in the definition of beta which is in the definition of gamma. Where this is helpful, it is also possible that the expression in which the error was found will occur in as the first item in the position list, for example

incorrect letrec form
    in (letrec a . b) in alpha

The ⟨position⟩ may also be

in the body of the program

which represents the body (third component) of a lambda-expression, or the body (second component) of a let— or letrec-expression, possibly nested in the body position of outer lambda-, let— or letrec-expressions.

The checker announces itself, by way of prompting for its only input, which is the expression to be checked. The subsequent output is then either a list of errors, or is

revealed no errors

in the case of a syntactically correct expression. Accordingly, an interaction might proceed as follows:

might proceed as follows:

```
Take input from where? SYNTAX.LOB
Syntax check
Take input from where? MAP_UNTIL_END.LSO
load_code used but not defined
    in the body of the expression
newline used but not defined
    in map_until
```

The syntax checker may also be used to list the names of the unbound variables of an expression, since each occurrence of each of these will be reported as an error. In this particular example, both of the errors reported are of this form, since MAP_UNTIL_END.LSO contains these two references to the library STANDARD.LIB.

Construction notes

SYNTAX.LSO requires the following libraries

```
SYNTAX_FUNCTION.LIB
SYNTAX_ERROR.LIB
LIPKIT.LIB
S_EXPRESSION.LIB
STANDARD.LIB
SECD_CODE.LIB
```

and the composite text compiles with no linking to form the program SYNTAX.LOB.

The list structure editor

The list structure editor is a tool for constructing, inspecting and modifying s-expressions, which are the visible representations of the data objects in the LispKit system. The editor operates on a single expression, which is referred to in this section as *the file*. Normally, the file would be read in from a filing system file, hence the name, although a new file may be created by typing its value at the beginning of an editing session.

The session proceeds by the user typing a sequence of commands, which are acted upon by the editor. At any time, there is a distinguished sub-expression of the file which is called *the current expression*. This is the part of the file on which the user's attention is focused at the time. There are commands for displaying the current expression, commands for navigating about the file by changing which particular sub-expression is the current one, and various commands for making changes to the current expression. At any time during the session, the file may be written out to the output stream, and so to a filing system file. This mechanism may be used either for checkpointing the edit, or to save the final result of an edit.

Each command will be described in turn, and an outline of an editing session follows. It is convenient first to introduce the notions of component and of pattern matching.

Components

Various of the commands in the editor operate on the components of the current expression. For the purposes of this editor, the (immediate) components of a list such as

```
(un (dau (tri) . pedwar) (pump . chwech) saith)
```

are its head, the head of its tail, the head of the tail of its tail, ···

```
un
(dau (tri) . pedwar)
(pump . chwech)
saith
```

In the editor, they are numbered from one, so that component three of the above list is

```
(pump . chwech)
```

For the purposes of the editor, it is convenient to extend the notion of component to general s-expressions which may not lists, because they do need not end at a final **NIL**. Thus

```
(un (dau (tri) . pedwar) (pump . chwech) saith . wyth)
```

has a fifth component, which is the atom

    wyth

By this definition, the last component of a list is always NIL.

Pattern matching

A pattern is an s-expression which is given a meaning by the process of matching it against an expression. A pattern matches an expression: if the pattern is simply a number; or if the pattern is a symbolic atom and is equal to the expression; or if the components of the pattern match corresponding parts of the expression. That is, in order to match, the pattern must be the same as the expression, except that a number in the pattern may stand for any s-expression in the corresponding position in the expression, provided that two occurrences of the same number stand for equal sub-expressions of the expression. The pattern

    unigryw

matches only one expression, namely

    unigryw

The pattern

    99

matches any s-expression. The pattern

    (un.dau)

again matches only one expression,

    (un.dau)

The pattern

    (Peredur (Rhonabwy.1) .1)

matches many expressions, including each of

    (Peredur (Rhonabwy))
    (Peredur (Rhonabwy.x) .x)
    (Peredur (Rhonabwy cyntaf ail) cyntaf ail)

A template has the same form as a pattern, and is always used in conjunction with a pattern. When a pattern is found to match a given expression, the matching defines a correspondence between the numbers which occur in the pattern, and the sub-expressions of the expression which each number represents. *An instance of the template in this environment* is the expression obtained by substituting for the numbers in the template, replacing them by the corresponding sub-expressions of the matched expression. Thus, in an environment where the pattern

    (Peredur (Rhonabwy.1) .1)

matches the expression

(Peredur (Rhonabwy Macsen Wledig) Macsen Wledig)

the value of the template

(Rhonabwy (Peredur 3) (1.1))

is the expression

(Rhonabwy (Peredur 3) ((Macsen Wledig) Macsen Wledig))

Display and output commands

At any stage during the session, a print command may be issued which displays the current expression. The editor will never spontaneously show the current expression, so print commands are used frequently. The full form of the command is either

(p all)

which outputs the whole of the current expression, or

(p ⟨n⟩)

where ⟨n⟩ is a positive integer. This latter form outputs only the top ⟨n⟩ levels of the structure of the current expression, and any fine structure embedded in more than ⟨n⟩ parentheses will be suppressed, being replaced by an asterisk. In each case, the displayed expression is formatted, by the insertion of line breaks and indentation which reflect the structure of the expression.

For convenience, the most frequently used form of the command is available as an abbreviation, which is simply the atom

p

and which stands for the command

(p 3)

Irrespective of the size of the current expression, this gives sufficient detail to plan the next command or two, and it normally produces a displayed expression which is small enough that its structure can rapidly be assimilated.

Because the formatting of p-output requires a great deal of computation for large expressions, and because the formatting would greatly increase the size of files, it is undesirable always to format the output which is sent to disk files. Accordingly, there is a command

(p file)

abbreviated to

file

which outputs the current expression without any formatting. This command is intended to be used only for writing to filing system files.

Navigation commands

Each navigation command selects a new current expression. It does not make any change to the file.

Provided that the current expression is not an atom, its $\langle n \rangle$th component may be selected as the new current expression by typing the number

$\langle n \rangle$

as a command. If there are less than $\langle n \rangle$ components in the current expression, then the editor will report this as an error.

More generally, the find command may be used to select a sub-expression which matches a pattern. The find command

(f $\langle$pattern$\rangle$)

selects a sub-expression of the current expression which matches the *⟨pattern⟩*. In the event that there is more than one matching sub-expression, the new current expression is the first match found by searching the current expression in component order. The component order of searching an expression $\langle e \rangle$ is: firstly, the immediate components of $\langle e \rangle$, from left to right; then a component order search of each of the immediate components of $\langle e \rangle$ taken in turn. Generally speaking, the find command is only used to match an immediate subcomponent, or to find the unique occurrence of a pattern in the current expression. If there are no matches for a pattern, the find command is reported as an error.

As the user navigates through the file, moving the current expression deeper into the structure of the file, the editor keeps a record of the upper levels of the structure which are no longer a part of the current expression. This record is the *context* of the current expression. The up command,

u

restores one layer of the context to the current expression, having the effect of moving one level up the structure tree. Thus an up command immediately following a successful

$\langle n \rangle$

command undoes the effect of that command. After a successful find, an up command still moves up by only one level of structure, so may or may not reverse the effect of the find command, depending on the depth to which the find had to search before finding a match.

The whole of the context may be restored to the current expression by the top command,

top

Since the file consists of the current expression and its context, the top command has the effect of selecting the whole of the file as the new current expression.

File modification commands

The commands which make changes to the file all act by making changes to the current expression, without changing the context. If a change is made to the current expression, and the context restored to the expression by moving up, then it is the changed expression which becomes a part of the new current expression, and so of the new file.

Provided that the current expression is not an atom, the after and before commands

(a ⟨*pattern*⟩ ⟨*template*⟩)   and   (b ⟨*pattern*⟩ ⟨*template*⟩)

identify the first immediate component of the current expression which matches the ⟨*pattern*⟩, and insert an instance of the ⟨*template*⟩ as a new immediate component either after, or before, that matched component. It is an error to attempt either command when there is no immediate component which matches the ⟨*pattern*⟩.

Complementary to the after and before commands, the delete command removes a component from the current expression. Provided that the current expression is not an atom, the command

(d ⟨*pattern*⟩)

removes from the current expression the first immediate component, reading from left to right, which matches the ⟨*pattern*⟩. There is an error if the current expression has no components which match the pattern. There is a convenient abbreviation

d

which deletes the first component of the current expression.

Wholesale changes to the structure of the current expression are made with the change (or construct) command

(c ⟨*pattern*⟩ ⟨*template*⟩)

which attempts to match the whole of the current expression. If successful it replaces the current expression by an instance of the ⟨*template*⟩ in that environment. There is an error if the ⟨*pattern*⟩ does not match the current expression.

The replace command may be thought of as a particular degenerate case of the change command. The command

(r ⟨*expression*⟩)

replaces the whole of the current expression by the new ⟨*expression*⟩.

The exchange command operates on the immediate components of the current expression in exactly the way that the change command operates on the whole of the current expression. If the current expression is not an atom, the command

(e ⟨*pattern*⟩ ⟨*template*⟩)

replaces the first immediate component which matches the ⟨*pattern*⟩ by an

instance of the ⟨*template*⟩ in that environment. There is an error if there is no such component.

Each of the modifiers described so far makes only one change to the current expression. The global change command

    (g ⟨pattern⟩ ⟨template⟩)

finds all component lists of the current expression which match the ⟨*pattern*⟩ and replaces them by the corresponding instances of the ⟨*template*⟩. Notice that the replacement process acts recursively on the parts of the original expression substituted into the ⟨*template*⟩. Because the process could not terminate. it is an error to attempt a global change of the form

    (g ⟨n⟩ ⟨template⟩)

where the ⟨*n*⟩, being a number, would match the whole of the current expression. It is not an error to perform a global change with a pattern that matches no component lists of the current expression.

The final, and most important of the changing commands is the backtracking command,

    undelete

which will undo the effect of any one of the changes, if performed immediately after that change. Its effect is to replace the current expression by the expression which was current immediately before the most recent change command. Thus, an expression may be changed by any one of the change commands, and if the result is unsatisfactory, then the change may be undone by an undelete command.

The undelete command is of course itself a change command, and so may itself be undone by an undelete command. The undelete command may also be used to move portions of a file about, since the value which is restored by an undelete command is not affected by any intervening combination of movement and printing commands.

The end command

Perhaps the most useful command that may be issued to any program is the command which stops that program. The command

    end

is used to end an editing session. Notice that it does not write anything to any filing system file, and that the whole of the editing session is lost after the end command, unless some positive action was taken to preserve it earlier in the session. The normal way to finish a successful editing session is therefore

⟨ctrl Y⟩⟨return⟩	to redirect the output
Send output to where? ⟨file name⟩	file to receive edited expression
top file	output the whole of the file
⟨ctrl Y⟩⟨return⟩	to close the output file

Send output to where? *CONSOLE:*
end   to leave the editor
Exit editor

An example of an editing session

The script below is that of a session which changes the error message from
the editor. Notice in particular: that any number of commands may be
typed on a line; that there is no output from the editor unless specifically
requested; that the saving of the changed file must also be explicitly
requested; that the whole of the file should be selected by a top command
before the file command if the whole file is to saved.

Take input from where? *E.LOB*   to execute the editor
Editor
Take input from where? *E_CONTROL.LIB*   the text to be edited
ready

Take input from where? *CONSOLE:*   the source of the editing
                                                  commands
p   the first command

((edit lambda (f i) (editloop i *))   output by the editor
  (editloop lambda (i t) (letrec . *))
  (editstep lambda (c t) (let . *)))

(l (quote Error)) u p   three separate commands

(list newline (quote Error))

(e (quote Error) (quote NumbSkull!!)) p   the change being made

(list newline (quote NumbSkull!!))

⟨*ctrl Y*⟩⟨*return*⟩   to save the resulting file
Send output to where? *ABRUPT_E_CTRL.LIB*
top file   write out the whole of the file
⟨*ctrl Y*⟩⟨*return*⟩   close the output file
Send output to where? *CONSOLE:*
end   and leave the editor
Exit editor

Construction notes

E.LSO requires the following libraries

```
E_CONTROL.LIB
E_FUNCTION.LIB
E_MATCH.LIB
E_MISC.LIB
S_EXPRESSION.LIB
STANDARD.LIB
SECD_CODE.LIB
```

and the composite text compiles with no linking to form the program E.LOB.

The LispKit Lisp library manager

The library manager is a utility for constructing a complete LispKit Lisp program from an outline of the program and a number of libraries which contain the component functions of the program. A library is an association list, like the definitions in the tail of a LispKit letrec – form. The LispKit system contains, for example, a library of standard functions which has the form ---

```
( (append.(lambda (e1 e2) ···))
  (member.(lambda (e l) ···))
  (equal.(lambda (e1 e2) ···)) ···)
```

and so contains definitions of the functions append, member, equal, amongst others. The library manager takes a LispKit Lisp expression which may have free variables, and then takes a number of libraries, extracting definitions of the free variables from those libraries, and assembling the whole into an expression with no free variables. Only expressions with no free variables are acceptable to the compiler.

For illustration, take the example of the library manager itself, which (barring a bootstrapping problem!) was written using the library manager. The text of the program is in the file LIBMAN.LSO, and is a LispKit Lisp expression, of the form

```
(letrec (lambda (i) (append ··· (librarian ···)))
        (librarian ···)
        (bind ···) )
```

in which a number of the variable names which appear are unbound, for example append, and freevars. Now freevars is defined in a library called LISPKIT.LIB, which contains functions that are connected with the compilation of LispKit Lisp programs. If this library is presented to the library manager, it constructs the expression

```
(letrec (letrec (lambda (i) (append ··· (librarian ···)))
                (librarian ···)
                (bind ···) )
        (freevars ···) )
```

Some of the functions, such as append, are still free in this expression, and so also is a new function, addelement, used in the definition of freevars, and defined in the library SET.LIB. Presented with this library next, the manager constructs an expression of the form

```
(letrec (letrec (letrec (lambda (i) (append ⋯ (librarian ⋯)))
                        (librarian ⋯)
                        (bind ⋯) )
                (freevars ⋯) )
        (addelement ⋯)
        (union ⋯)
        (intersection ⋯) ⋯)
```

thus adding, for each library, an extra level of letrec — definition to the outside of the program being constructed.

Finally, if this is necessary, the manager encloses the expression in a further level of definition so as to catch all free occurrences of the standard functions that the compiler recognises as operators. Thus, for example, in the expression

```
(head (map head (quote (a.b) (c.d))))
```

the first occurrence of head is recognised by the compiler, and so is not considered to be a free variable of the expression, but the second occurrence is free. If this expression were passed through the manager, along with a library defining the function map then the result would be

```
(let (letrec (head (map head (quote (a.b) (c.d))))
            (map. ⋯) )
    (head. (lambda (arg1 arg2) (head arg1 arg2)) ) )
```

where the second occurrence of head is now a reference to the function defined at the outside level.

When the expression has been closed, by defining all of its free variables, the manager writes the complete expression, so that it may subsequently be submitted for compilation.

At each stage of the construction process, the manager prompts for a new library by listing the free variables of the current expression, so the interaction to construct the library manager itself would proceed as follows

Take input from where? *LIBMAN.LOB*   program to be executed
LispKit Librarian
Take input from where? *LIBMAN.LSO*   skeleton of expression
sequence newline filter close reduce member union list of free variables
difference map append freevars
Take input from where? *LISPKIT.LIB*   first library
addelement sequence newline filter close reduce
member union difference map append
Take input from where? *SET.LIB*   second library
sequence newline filter close reduce member map
append
Take input from where? *STANDARD.LIB*   third and final library
Type anything to print result
Take input from where? *CONSOLE:*   to permit output redirection
⟨*ctrl Y*⟩⟨*return*⟩
Send output to where? *WORKFILE*   destination for result
file   causes output to be sent
⟨*ctrl Y*⟩⟨*return*⟩   closes output file

At any stage in the construction process, if the reply to the prompt of a list of free variables is not a library, but the atom

    end

then the manager will immediately allow the output of the expression in its present, uncompleted state; if the answer is the atom

    abort

then the library manager terminates immediately, with no output.

The right hand sides of the definitions in a library may be recursive, in that they refer to their own left hand sides; they may make reference to other definitions in the same library; they may also contain references to definitions in other libraries. In this last case, the library containing the reference must be offered for binding before the library containing the definition; notice that this requirement implies that mutually recursive definitions must both occur in the same library. Libraries which are irrelevant to the expression, because they contain no definitions of variables free in the expression, are simply ignored, and the same prompt is repeated.

Whenever a program in the distribution suite is constructed by use of the library manager, the construction notes in the section of the manual relating to that program contain a list of the libraries which are required, in the order in which they should be supplied to the library manager.

Construction notes

LIBMAN.LSO requires the following libraries

    LISPKIT.LIB
    SET.LIB
    ASSOCIATION.LIB
    STANDARD.LIB
    SECD_CODE.LIB

and the composite text compiles with no linking to form the program LIBMAN.LOB.

The bootstrap loader and run time system

The bootstrap loader and run time system, LOADER, is normally the first program loaded into the LispKit virtual machine. It successively loads and executes the codes of other LispKit Lisp programs until the machine is halted. The majority of the loader is written in LispKit Lisp, but there are a handful of applications of pseudo-functions. It is not possible to apply the methods used to reason about functional programs to these, since their evaluation causes side-effects. The coding of the loader should be compared with 'systems programming' in more conventional high-level languages, where it is also necessary to write unusual code in order to manipulate the interface with the underlying machine. Compared with embedding the run time system in the virtual machine, this approach makes the implementation more readily portable, and facilitates reasoning about the gross behaviour of the run time system.

The body of the letrec-form which is the loader,

```
(run_and_halt load_go_loop)
```

does not have a value, nor does any attempt to evaluate it ever terminate. The argument

```
load_go_loop
```

is a proper expression, whose value is a pair consisting of a function and a list. This value represents the application of the function to the list, as an argument list. In the course of executing the loader the application is evaluated, and this evaluation causes each successive program to be executed in turn. In this sense, the loader evaluates each program for the side-effect of reading the input to and writing the output from each of these programs. Since no LispKit Lisp expression has any such side-effect, it follows that the parts of the loader which accomplish the reading and writing are not written in LispKit Lisp.

The ordinary user need not understand the behaviour of the loader in any detail; he need only know that the code of the loader normally resides in a well known place in the filing system of his machine, and that the LispKit virtual machine reads this code at the beginning of every programming session. Throughout this manual, it is assumed that there is a copy of the closure from LOADER.CLS in this bootstrap file. If the machine is unable to find the bootstrap file, then it will output a message to that effect, for example, on the Perq the message is

```
No file SECD.BOOT
```

and is followed by a prompt for the name of a file which does contain a

bootstrap loader. In this way, a sophisticated user may choose to replace the loader which is normally used by the LispKit system. Details of the implementation of the loader are available, for those who are interested, in the implementor's guide, towards the end of this manual.

Related to the loader is the short program HALT which can be executed as an ordinary LispKit program, but which causes the loader and virtual machine to terminate, and returns control to the host operating system.

Construction notes

LOADER.LSO requires the following libraries

    S_EXPRESSION.LIB
    STANDARD.LIB
    SECD_CODE.LIB

and the composite text compiles with no linking to form the closure in LOADER.CLS.

The end of session program

Complementary to the bootstrap loader, LOADER, the halting program HALT is run to end a LispKit session. The program is another non-Lisp application of the form

```
(run_and_halt (quote NIL))
```

which the machine recognises as a special case. The effect of executing this program is to cause the machine to terminate successfully, and to return control to the host operating system. On some machines, this may involve re-booting the host.

Construction notes

HALT.LSO requires the library

SECD_CODE.LIB

and the composite text compiles with no linking to give the code in **HALT.LOB**.

Support for independent compilation

The LispKit system provides for components of a LispKit Lisp program to be compiled separately from each other, and for the various codes so produced to be loaded together to form a single composite code equivalent to that which would be compiled from the source of the whole of the program. This technique is employed in the system itself, (for example, the bootstrap loader is compiled separately from user programs) and is also available to users for use within their own programs. It is useful both for decomposing large programs into readily manipulated component parts, and for constructing new programs from pre-existing building blocks.

Details of the implementation of separate compilation are to be found in the implementor's guide. In essence, what the system provides is a mechanism for reading code objects from the input stream, and a function

load_code

in the STANDARD library which transforms these codes into the values which they represent. This is most readily explained by example. Consider the program

```
(let (lambda (input) (f g input))
        (f. (lambda (x i) (cons x (cons x (cons (head i) (quote NIL)))) ))
        (g. (quote Hello)) )
```

This could be compiled and executed as a single entity, as follows. Assume that its source text is in the file TOGETHER.LSO.

```
⟨ctrl Y⟩⟨return⟩
Send output to where? TOGETHER.LOB
⟨ctrl Z⟩

Take input from where? LC.LOB

Take input from where? TOGETHER.LSO

Take input from where? CONSOLE:
⟨ctrl Y⟩⟨return⟩
Send output to where? CONSOLE:
⟨ctrl Z⟩

Take input from where? TOGETHER.LOB
Hello Hello
Take input from where? CONSOLE:
hello
hello
```

Now, if it was desired to treat the definitions of f and g separately from the body of the program, the following three text files would be created

```
LOAD2.LSO
        (lambda (input) (let (f g real_input)
                            (f. (load_code (head input)))
                            (g. (load_code (head (tail input))))
                            (real_input. (tail (tail input)))) )
F.LSO
        (lambda (x i) (cons x (cons x (cons (head i) (quote NIL)))) )
G.LSO
        (quote Hello)
```

and they would then be compiled separately. Notice that the filename convention used for the system files is that names with the .LOB extension are for files containing the codes of stream to stream functions only, and that other codes are kept in files with the extension .CLS.

The following script describes the construction of a complete expression from LOAD2 and the STANDARD library, which defines the function load_code.

```
Take input from where? LIBMAN.LOB
LispKit Librarian
Take input from where? LOAD2.LSO
load_code
Take input from where? STANDARD.LIB
Type anything to print result
Take input from where? CONSOLE:
⟨ctrl Y⟩⟨return⟩
Send output to where? WORKFILE
file
⟨ctrl Y⟩⟨return⟩
```

The three expressions which are the separate components of the new program would be compiled

```
Send output to where? LOAD2.LOB
⟨ctrl Z⟩

Take input from where? LC.LOB

Take input from where? WORKFILE

Take input from where? CONSOLE:
⟨ctrl Y⟩⟨return⟩
Send output to where? F.CLS
⟨ctrl Z⟩

Take input from where? LC.LOB

Take input from where? F.LSO
```

```
Take input from where? CONSOLE·
⟨ctrl Y⟩⟨return⟩
Send output to where? G.CLS
⟨ctrl Z⟩

Take input from where? LC.LOB

Take input from where? G.LSO

Take input from where? CONSOLE:
⟨ctrl Y⟩⟨return⟩
Send output to where? CONSOLE:
```

and to run the program, all that is required is to execute the LOAD2 object code with an input stream which begins with the objects for f and g

```
⟨ctrl Z⟩

Take input from where? LOAD2.LOB

Take input from where? F.CLS

Take input from where? G.CLS
Hello Hello
Take input from where? CONSOLE:
hello
hello
```

This method of execution is suitable whilst a program is being developed, but once all the parts of a separately compiled program have been completed, it is convenient to be able to disguise the separateness of the components from a user of the program. This linking is simply done by concatenating the component object codes of the program into a single file, thus

```
⟨ctrl Y⟩⟨return⟩
Send output to where? APART.LOB
⟨ctrl Z⟩

Take input from where? CONCAT.LOB

Take input from where? LOAD2.LOB

Take input from where? F.CLS

Take input from where? G.CLS

Take input from where? CONSOLE:
end
⟨ctrl Y⟩⟨return⟩
Send output to where? CONSOLE:
```

The composite object code may now be executed in exactly the same way as the original code of TOGETHER

⟨*ctrl Z*⟩

Take input from where? *APART.LOB*
Hello Hello
Take input from where? *CONSOLE:*
hello
hello

since each of the component objects automatically appears in the input stream where they are required.

A number of general purpose loaders, similar to LOAD2, are provided as parts of the LispKit system, and are used as components of some of the system utilities. Descriptions of these programs follow.

The loader COMPOSE

The loader **COMPOSE** takes a sequence of object codes for functions of one argument. It loads these codes in such a way that the composition of the functions is applied to the first item in the input stream, and the result is written as the sole component of the output stream.

In order to use COMPOSE, its input stream must contain a number $\langle n \rangle$, followed by $\langle n \rangle$ object codes, each for a function of one argument, and then the one argument, to which the composition of the functions is applied. As an example, consider the LispKit Lisp summariser, STRUCTURE, which is composed from two functions of one argument,

```
structure = (lambda (s) ⟨skeleton of s⟩)
pretty = (lambda (e) ⟨formatted copy of e⟩)
```

The summariser output consists of a formatted skeleton of the first item in its input stream, so to run the summariser, the input stream should contain in turn

the code for the loader	COMPOSE.LOB
the number of composed codes	2
the codes for the functions	PRETTY.CLS
	STRUCTURE.CLS
the text of the argument	⟨expression⟩

accordingly, the file **STRUCTURE.LOB**, which contains the code that is normally executed to display the bindings of an expression, consists of the four items

the code for the loader	COMPOSE.LOB
the number of composed codes	2
the codes for the functions	PRETTY.CLS
	STRUCTURE.CLS

and expects the next item in the input stream to be the expression being analysed. This file behaves as though it contained the code of

```
(lambda (input) (cons (pretty (structure (head input))) (quote NIL)))
```

Construction notes

COMPOSE.LSO requires the library

STANDARD.LIB

and the composite text compiles with no linking to form the code **COMPOSE.LOB**.

The loader LOADK

The loader LOADK (κ for constant) is intended for use with constant expressions which define an output stream but require no arguments. It loads the code of a constant expression into an environment which ignores the input stream.

In order to use LOADK, its input stream must begin with the code of such a constant expression. As an example, the expression

```
(letrec (first (quote 10) ones)
   (ones. (cons (quote 1) ones))
   (first. (lambda (n l)
      (if (eq n (quote 0)) (quote NIL)
            (cons (head l) (first (sub n (quote 1)) (tail l)))) )) )
```

whose value is a list of ten ones, could be compiled to give a closure which would then be used as the argument to LOADK, and which might be called, say TEN_ONES.CLS. The input stream should contain, in order

the code for the loader	**LOADK.LOB**
the code for the expression	**TEN_ONES.CLS**

If desired, the codes could be linked to form a program which always output a list of ten ones, by concatenating these two codes into a single .LOB file. Such a file would behave as though it contained the code for

```
(lambda (input) ten_ones)
```

Construction notes

LOADK.LSO requires the library

STANDARD.LIB

and the composite text compiles with no linking to form the code **LOADK.LOB.**

The loader LOADS

The loader LOADS (S for standard) is intended for use with expressions which were written to run on the virtual machine described in the book. The programs for that machine expect to receive a fixed finite number of arguments, and deliver a single, possibly atomic, result. LOADS loads the code of such an expression into an environment which reads the arguments from the input stream, and delivers the result as the only component of an output stream.

In order to use LOADS, its input stream must contain the code of a function of $\langle n \rangle$ arguments, followed by the number $\langle n \rangle$, and then the arguments, in sequence. As an example, consider the LispKit Lisp compiler, LC, which is a function of one argument

```
compile = (lambda (e) ⟨make code for e⟩)
```

This has been compiled to produce the closure in LC.CLS, which expects a single argument and returns the compiled code. In order to use this code, the following sequence must appear in the input stream

the code for the loader	LOADS.LOB
the closure for LC	LC.CLS
the number of arguments	1
the argument	⟨source expression⟩

accordingly, the file LC.LOB, which contains the code that is normally executed to perform a compilation, consists of the first three of these items

the code for the loader	LOADS.LOB
the closure for LC	LC.CLS
the number of arguments	1

and expects the next item in the input stream to be the source expression to be compiled. This file behaves as though it contained the code for

```
(lambda (input) (cons (compile (head input)) (quote NIL)))
```

Construction notes

LOADS.LSO requires the following libraries

```
STANDARD.LIB
SECD_CODE.LIB
```

and the composite text compiles with no linking to form the code LOADS.LOB.

The loader
MAP_UNTIL_END

The loader MAP_UNTIL_END takes the object code for a function of one variable, and applies this function to each of the items in its input stream, up to but excluding the first occurrence of the atom *end*. The output stream which it constructs consists of the sequence of results of these applications, in the order of the arguments in the input stream.

In order to use MAP_UNTIL_END, its input stream must contain the code of a function of one argument, followed by the sequence of arguments to which this is to be applied, and terminated by the atom *end*. As an example, consider the s-expression formatter, PRETTY, which is a function of one argument

pretty = (lambda (e) ⟨formatted copy of e⟩)

This has been compiled to produce the closure in PRETTY.CLS, which expects a single argument and returns the formatted form. In order to use this code, the following sequence must appear in the input stream

the code for the loader	MAP_UNTIL_END.LOB
the closure for PRETTY	PRETTY.CLS
a number of arguments	⟨e 1⟩
	⟨e 2⟩
	...
	⟨e n⟩
the terminator	end

If a file was constructed, containing the two items

the code for the loader	MAP_UNTIL_END.LOB
the closure for PRETTY	PRETTY.CLS

then this file would behave like the code of the program

(letrec (lambda (input) (map pretty (untilend input))))

Construction notes

MAP_UNTIL_END.LSO requires the library

STANDARD.LIB

and the composite text compiles with no linking to form the code MAP_UNTIL_END.LOB.

The linker CONCAT

One of the programs central to the separate compilation scheme is the linker, CONCAT, which is used to concatenate a number of s-expressions into a single file. Its function is to copy a number of expressions from the input stream to the output stream, terminating at the first occurrence of the atom *end*.

Construction notes

There is no text for the program CONCAT. The code in the file CONCAT.LOB is formed by linking

MAP_UNTIL_END.LOB
IDENTITY.CLS

where **IDENTITY.CLS** contains the closure for the function

identity = (lambda (x) x)

Formatting and printing utilities

When being manipulated in the machine, complex data structures are represented by areas of the store and by values which are machine addresses of the host machine. There is no information in this structure about the layout of any s-expressions which were input to the machine. If you look at the output of a program such as the library manager, you will see that any useful layout has been removed, principally to economise on storage space. It is exceedingly difficult to see the structure of an s-expression presented in this form. A number of programs have been provided which assist in the reading of large s-expressions.

PRETTY is a program which lays out an s-expression so that the indentation reflects the nested structure of the expression. DUMP is a program that obscures (dumps) the fine detail of an s-expression, leaving its overall structure more apparent. A number of programs, such as the editor, E, make use of the pretty printer and summariser to format the interaction with the terminal. STRUCTURE and SHOW_LIB are programs specific to LispKit Lisp, which take a LispKit Lisp source expression, and display the skeleton which supports the structure of the expression, and label the places where each new name is introduced.

An s-expression pretty printer

The s-expression pretty printer produces its output by assembling a copy of the first item in its input stream, into which copy it inserts atoms whose printed representations are layout characters. This is done in such a way that the layout of the printed result reflects the nesting structure of the input expression.

If an expression is sufficiently small that it will fit onto a line, then it is printed in that form; if not, then each of its components is formatted individually, and they are laid out in a vertical column, slightly indented. The pretty printer recognises structures which are tail nested in similar structures, for example the else-if pair in

$$(\text{if} \langle c\ 1 \rangle\ \langle t\ 1 \rangle\ (\text{if} \langle c\ 2 \rangle\ \langle t\ 2 \rangle\ \langle e\ 2 \rangle))$$

and lays these out in a vertical column, instead of increasing the indentation for the inner expression.

The source listings in the companion volume to this manual were each produced with this pretty printer.

Construction notes

There is no filed source text for the body of the program, which was constructed by the library manager from the expression

    pretty

(which has one free variable) and the libraries

    S_EXPRESSION.LIB
    STANDARD.LIB
    SECD_CODE.LIB

and the composite text compiled to give the code in **PRETTY.CLS**. The code in **PRETTY.LOB** is formed by linking

the closure	**LOADS.LOB**
the closure	**PRETTY.CLS**
the argument count	1

so that the resulting code calculates

    (lambda (kb) (cons (pretty (head (kb))) (quote NIL)))

An s-expression summariser

The s-expression summariser produces its output by assembling a copy of the first item in its input stream, but omitting from the copy any substructure of the input expression which would be nested in more than two parentheses in the output. An asterisk is substituted for the omitted detail. The resulting expression is normally of a size that its structure can be readily seen.

As an example, consider the result of presenting the library S_EXPRESSION.LIB as input to the summariser

Take input from where? *DUMP.LOB*

Take input from where? *S_EXPRESSION.LIB*
((dump let . *) (flatten letrec . *) (pretty let . *))

The library is readily seen to be a list of three definitions. and they are seen to be definitions of dump, of flatten, and of pretty.

Construction notes

There is no filed source text for the body of the program, which was constructed by the library manager from the expression

dump

(which has one free variable) and the libraries

S_EXPRESSION.LIB
STANDARD.LIB

and the composite text compiled to give the code in DUMP.CLS. The code in DUMP.LOB is formed by linking

the closure	LOADS.LOB
the closure	DUMP.CLS
the argument count	2
the first argument	2

so that the resulting code calculates

(lambda (kb) (cons (dump (quote 2) (head kb)) (quote NIL)))

The LispKit Lisp summariser

The summariser displays the point of declaration of each of the names defined in a LispKit Lisp expression. It produces its output by assembling a copy of the first item in its input stream, but eliding from the copy all expressions which define no new names. The resulting output expression consists of a skeleton of name-defining constructs, in which the bodies of the expressions are replaced by asterisks. This output is formatted according to the criteria used by the pretty printer described above.

As an example, consider the summary of the library manager, LIBMAN, obtained by applying STRUCTURE to the expression in the file LIBMAN.LSO by the following interaction

```
Take input from where? STRUCTURE.LOB

Take input from where? LIBMAN.LSO
( letrec
  ( lambda ( i ) * )
  ( librarian
    lambda
    ( e u i )
    ( letrec
      *
      ( missing . * )
      ( e' . * )
      ( u' . * )
      ( next . * )
      ( write lambda ( e i ) * ) ) )
  ( bind
    lambda
    ( e u a )
    ( letrec
      ( *
        *
        ( * * ( * ( lambda ( d ) * ) . * ) ) )
      ( defs * ( lambda ( d ) * ) . * )
      ( u'
        *
        ( lambda
          ( v )
          ( * ( lambda ( d l ) * ) . * ) ) . * )
      ( a'
        *
        ( lambda ( d ) ( * * ( lambda ( l ) * ) ) ) . * ) ) )
```

```
( bind_operators
  lambda
  ( e u )
  ( letrec
    *
    ( define
      lambda
      ( arity )
      ( lambda ( name ) ( let * ( arguments . * ) ) ) ) ) ) ) )
```

Every name introduced by a binding construct — let, letrec, or lambda —
in the expression in LIBMAN.LSO appears in the skeleton above. There
are no new bindings in any of the sub-expressions that have been
represented as asterisks in the skeleton. There are, of course, free variables
in LIBMAN.LSO which are references to libraries such as LISPKIT.LIB,
and the bindings for these are not shown, since they do not form a part of
the expression in LIBMAN.LSO.

Construction notes

There is no filed source text for the body of the program, which was
constructed by the library manager from the expression

```
structure
```

(which has one free variable) and the libraries

LISPKIT.LIB
STANDARD.LIB
SECD_CODE.LIB

and the composite text compiled to give the closure in STRUCTURE.CLS.
The code in STRUCTURE.LOB is formed by linking

the closure	COMPOSE.LOB
the function count	2
the closure	PRETTY.CLS
the closure	STRUCTURE.CLS

so that the resulting program calculates

```
(lambda (kb) (cons (pretty (structure (head kb))) (quote NIL)))
```

The library summariser

The library summariser is a small extension of the LispKit Lisp expression summariser, and displays the introduction of each of the names defined in a LispKit Lisp source library. It produces its output by assembling a copy of the first item in its input stream, but eliding from the copy all expressions which define no new names. The resulting output expression consists of a skeleton of name-defining constructs, in which the bodies of the expressions are replaced by asterisks. This output is formatted according to the criteria used by the pretty printer described above.

The result is an expression which lists the names of each of the definitions included in the library, together with a skeleton of the definition given for that name.

Construction notes

The text in SHOW_LIB.LSO compiles to yield the closure in SHOW_LIB.CLS. It is a function which converts a library into a letrec-expression which can then be dealt with by the summariser. Accordingly, the code in SHOW_LIB.LOB is formed by linking

the closure	COMPOSE.LOB
the function count	3
the closure	PRETTY.CLS
the closure	STRUCTURE.CLS
the closure	SHOW_LIB.CLS

so that the resulting program calculates

```
(lambda (kb)
    (cons (pretty (structure (show_lib (head kb)))) (quote NIL)))
```

Some other LispKit utilities

In addition to the programs described so far, there are several programs, whose sources are listed in the companion volume to this, and which are parts of the LispKit system's programming environment. Some of these tools are alternative implementations of the functions of tools already described, others are definite extensions which we have found useful. In any case, these programs are included both for their own utility, and in order to guide the user in the design of his own programs.

An interpreter for LispKit Lisp

The interpreter provides an interactive environment for experimenting with the design of components of a LispKit Lisp program. Any valid LispKit Lisp expression may be evaluated by the interpreter, but additional checks are performed as compared with the normal execution of compiled code. This means that the interpreter provides a means of checking for programming errors at the expense of considerably slower execution. There is also some editing capability to facilitate experimentation.

Essentially, the interpreter prompts the user for a LispKit Lisp expression, evaluates that expression, writes the result, and repeats the prompt. Any valid LispKit Lisp expression may be evaluated in this way. In addition, expressions with free variables may be evaluated, provided that those free variables have been defined in an outer level environment which is manipulated by the various commands listed below.

If the user responds to the interpreter prompt with one of the special forms reserved for commands, then this is treated as a command, and not as an expression to be evaluated. Most of the commands manipulate an environment which is treated as the definition part of a letrec-form. Expressions evaluated by the interpreter are evaluated as though they appeared in the body position of this letrec-form. Notice particularly that command forms are only recognised when they appear as the response to a prompt; they are not treated specially when they appear as sub-expressions of another expression; they cannot be manipulated as if they were LispKit Lisp expressions.

Interpreter commands

Definitions may be added to the outer level environment by use of the command

    (def ⟨name⟩ ⟨expression⟩)

where ⟨name⟩ is an atom, and ⟨expression⟩ is a LispKit expression, possibly with free variables. This has the effect of binding the ⟨name⟩ to the value of the ⟨expression⟩, but notice that it is the value of the ⟨expression⟩ in the outer level environment that is in force when that ⟨name⟩ is used. This allows for mutual recursion between definitions; since definitions may be changed, this also means that each definition depends on the most recent definition of each of its free variables.

The interpreter maintains a stack of definitions for each defined name, so that successive uses of the *def* command will mask previous definitions. The complementary command

    (cancel ⟨name⟩)

discards the most recent binding of the ⟨*name*⟩, making the most recent preceding definition (if any exists) current. The list of defined names may be inspected at any time with the

vars

command which lists the names bound by each definition. Multiple occurrences of a name in the list indicate multiple definitions of that name.

The texts of definitions may be recalled by the *save* command

```
(save ⟨name 1⟩ ⟨name 2⟩ ⋯ ⟨name n⟩)
```

which was designed to save the definitions in a file. It writes to the output stream an expression which contains the texts of each of the definitions of each of the ⟨*name*⟩s listed, in the form

```
(restore (def ⟨name 1⟩ ⟨expression 1⟩)
         ...
         (def ⟨name n⟩ ⟨expression n⟩) )
```

and the interpreter recognises the command

```
(restore ⟨command 1⟩ ⋯ ⟨command n⟩)
```

as representing the sequence of listed commands. Thus the *save* command may be used to create a file which contains a *restore* list of the definitions in force in one interpreter session, and by including that file in the input stream of another session, those definitions may be reinstated. There is an abbreviation

save

which saves all of the definitions, specifically for transferring a complete session in this way.

The final command is

end

which is used to end an interpreter session. It does not preserve any of the state of the session, so this must be done by explicit saving if required.

Special names and forms

There are three names which are used by the interpreter in addition to the names of its commands. The first is a variable

patience

bound in the outer level environment. This should normally be bound to a small positive integer, and is prebound in this way at the beginning of the session. When the interpreter is evaluating an expression, it counts the number of symbols being written to the output stream and abandons the evaluation after the first patience number of symbols. This means, for example, that it is safe to attempt to print an infinite list.

```
      ⟩ patience
50
      ⟩ (quote (a b c d e))
( a b c d e )
      ⟩ (cancel patience)
patience cancelled
      ⟩ (def patience (quote 3))
patience defined
      ⟩ (quote (a b c d e))
( a b . . .
      ⟩ (letrec i (i. (cons (quote *) i)))
( * * . . .
```

The default value of patience is fifty, but you may find larger or smaller values appropriate in a particular case; in the above example, the value of three prevents the whole of any list of more than one element from being output.

The second special name is not properly a variable in the outer level environment, since it may not be defined by the user. The name

    it

is always bound to the value of the most recently evaluated expression, so that for example

```
      ⟩ (quote 2)
2
      ⟩ (add it it)
4
      ⟩
```

Notice particularly that it is not possible to use the it pseudo-variable to give a name to the result of an evaluation. The next example demonstrates this point

```
      ⟩ (quote cyntaf)
cyntaf
      ⟩ it
cyntaf
      ⟩ (def diweddaraf it)
diweddaraf defined
      ⟩ diweddaraf
cyntaf
      ⟩ (quote ail)
ail
      ⟩ diweddaraf
ail
      ⟩
```

This behaviour will be seen to be consistent with the behaviour of other free variables of definitions.

In addition to these names, there is a name which is reserved if it appears

at the head of an application. The form

```
(exec <closure>)
```

is a valid expression in the extension of LispKit Lisp which is recognised by the interpreter. The value of the expression in the *closure* position must have the same form as compiled code, that is it must be a pair, the head of which is a code sequence, and the tail of which is an environment. Further details of closures and code are to be found in the book, and in the implementor's guide.

The value of an *exec* expression is the function for which the argument is the closure. The advantage of an *exec* expression is that its evaluation proceeds at the normal speed of compiled code, but at the disadvantage of evading all of the error checking of the interpreter. The closure of an *exec* expression may not refer to any of the definitions in the top level environment.

```
> (def greet
       (lambda (who) (cons (quote Hello) (cons who (quote NIL)))))
greet defined
> (greet (quote world))
(Hello world)
> (def fastgreet
       (exec (quote ((2 NIL 1 (0.0) 24 13 2 Hello 13 5).NIL))))
fastgreet defined
> (fastgreet (quote world))
(Hello world)
>
```

A description of the code may be found in the book, and in the implementor's guide in this manual.

An example session using the interpreter

```
Take input from where? INTERP.LOB
LispKit interpreter: type end to finish
>
Take input from where? CONSOLE:
(def pair (lambda (x y) (cons x (cons y (quote NIL)))))
pair defined
> (pair (quote A) (quote B))
( A B )
> <ctrl Y><return>
Send output to where? SAVEFILE
(save pair)
<ctrl Y><return>
Send output to where? CONSOLE:
(cancel pair)
pair cancelled
> vars
patience
> <ctrl Z>
```

Take input from where? *SAVEFILE*
> pair defined
>
Take input from where? *CONSOLE:*
(pair (quote 1) (quote 2))
(1 2)
> *end*
Exit interpreter

Construction notes

INTERP.LSO requires the following libraries

INTERP_MISC.LIB
TUPLE.LIB
STANDARD.LIB
SECD_CODE.LIB

and the composite text compiles with no linking to form **INTERP.LOB**.

An alternative s-expression editor

Like the list structure editor, this editor is tool for manipulating s-expressions, but this editor is of a considerably simpler construction, and has a smaller range of commands based on a simpler notion of s-expression. This editor has been largely superseded by the editor E.

The editor manipulates a single expression, called *the file*. An editing session proceeds by the user typing a sequence of commands. At any time there is a *current sub-expression* of the file, which indicates the focus of attention of the user. There are commands which select different current expressions, commands which display the current expression, and commands which make changes to the current expression.

The display command

At any stage during the edit session, a print command may be issued which displays the current expression. The editor will never spontaneously produce any output, excepting error messages, so print commands are used frequently. The full form of the command is either

   (p all)

which displays the whole of the current expression, or

   (p ⟨n⟩)

where ⟨*n*⟩ is a positive integer. This latter form displays only those parts of the expression which are enclosed in no more than ⟨*n*⟩ parentheses, replacing all deeper structures by a single asterisk, so as to make the overall structure more readily apparent.

For convenience, the most frequently used form of the command may be abbreviated to

   p

which is entirely equivalent to

   (p 2)

This gives sufficient detail to plan the next command, and normally displays an expression sufficiently small that its structure is readily visible.

The print command is also used to write a copy of the edited file to a filing system file, by redirection of the output stream, thus:

```
⟨ctrl Y⟩⟨return⟩
Send output to where? ⟨output file⟩
top (p all)
⟨ctrl Y⟩⟨return⟩
Send output to where? CONSOLE:
```

Navigation commands

Each navigation command selects a new current expression; it makes no changes to the file.

Provided that the current expression is a pair, the head of the pair may be selected as the current expression by using the

h

command, and the tail may be selected by the

t

command. Thus a sequence of *h* and *t* commands may be used to select any sub-component of the current expression.

Whilst moving down through an expression, using the *h* and *t* commands, parts of the file pass out of the current expression. These parts and their relationship to the current expression form a *context*, which can be used by the editor to reconstruct the file from the current expression. The up command

u

undoes the effect of one *h* or *t* command, by recovering one layer of the file from the context. The whole of the file may be reselected as the current expression by use of the

top

command which restores the whole of the context to the current expression.

File modification commands

The commands which change a file all act on the current expression, without affecting the context. If a change is made to the current expression, and the eontext restored by *u* and *top* commands, then it is the changed expression which becomes a part of the file.

The fundamental changing command has the form

(c ⟨pattern⟩ ⟨template⟩)

which matches the structure of the ⟨*pattern*⟩ against the current expression. Pairs in the ⟨*pattern*⟩ must correspond to pairs in the expression, but each atom in the ⟨*pattern*⟩ can correspond to any sub-expression of the current expression. There is no check that multiple occurrences of an atom in the ⟨*pattern*⟩ correspond to the equal sub-expressions. If the match is successful, then the current expression is replaced by a copy of the ⟨*template*⟩, excepting that each occurrence of each atom present in the ⟨*pattern*⟩ is replaced by the expression to which it corresponds.

The replace commands may be thought of as being a particular special case of the change command, in that

(r ⟨expression⟩)

simply replaces the whole of the current expression by the new ⟨*expression*⟩.

To accommodate the human capacity for error, there is a command

> *undo*

which, if performed after any change, replace, or undo command, will undo the effect of that command, unless there have been any intervening navigation commands.

The end command

Perhaps the most useful command recognised by the editor is the command

> *end*

which terminates an editing session. Notice that this command does not save any of the results of the editing session, and that unless some positive action is taken to send the edited file to some filing system file, by using the mechanism of output redirection, the changes made during the session will be lost.

An example of an editing session

```
Take input from where? EDIT.LOB
Editor ready
Take input from where? E.LSO

Take input from where? CONSOLE:

p
( letrec ( lambda * * ) ( comment quote * ) ( edit lambda * * ) (editloop
lambda * * ) (editstep lambda * * ) )

t t h p
( comment quote ( * * ) )
t t h p
( ( List editor, Geraint ) ( last changed 21 February 1983 ) )
h t t p
( Geraint )
(c (x) (x Jones)) p
( Geraint Jones )
⟨ctrl Y⟩⟨return⟩
Send output to where? NEWFILE.LSO
(p all)
⟨ctrl Y⟩⟨return⟩
Send output to where? CONSOLE:
end
Exit editor
```

Construction notes

The text in EDIT.LSO is complete in itself, and compiles directly to give the code in EDIT.LOB.

An s-expression librarian

The librarian provides the ability systematically to insert common sub-expressions into any s-expression. It has been largely superseded as a tool for constructing LispKit Lisp source texts by the library manager LIBMAN which is described elsewhere in this manual.

The librarian takes an expression as input, and searches this expression for occurrences of sub-expressions of the form

    (include ⟨name⟩)

where ⟨name⟩ is an atom. Each of these is substituted for by a value obtained from the user in response to a prompt of the form

    ⟨name⟩ =

Normally, these responses would be taken from files. Only the first occurrence of a name causes the librarian to prompt for input; thereafter all occurrences of the same name are substituted for by the same value.

The substitution process is repeated on the substituted values until no includes remain. The librarian then gives the user the opportunity to redirect the output stream before writing the resulting expression to a file. Finally, the command

    end

terminates the program.

An example of a the use of the librarian

Take input from where? *LIBRARIAN.LSO*

Take input from where? *CONSOLE:*

(letrec (include body) (two.(include two)))

body = *(two (include list) (include list))*
two = *(lambda (a b) (cons a (cons b (quote NIL))))*
list = *(quote (a b c))*
Type 'end' to finish, anything else to print result
file
(letrec (two (quote (a b c))) (quote (a b c))) (two lambda (a b)
(cons a (cons b (quote NIL)))))
end
Exit librarian

Construction notes

LIBRARIAN.LSO requires the following libraries

TUPLE.LIB
ASSOCIATION.LIB
STANDARD.LIB
SECD_CODE.LIB

and the composite text compiles with no linking to form the object code
LIBRARIAN.LOB.

.

A simple variable scope checker

The scope checker is a program which analyses LispKit Lisp source expressions, and lists every occurrence of a variable not bound in that expression, together with a description of the position of the free occurrence in the expression. This function has been subsumed, for normal use, by the LispKit Lisp syntax checker, SYNTAX, which includes this check in its repertoire.

The checker takes a LispKit Lisp expression as the first item in its input stream, and outputs either the empty list of messages,

    NIL

to indicate a properly closed expression, or a list of messages, such as

    ((y f) (x g a))

to indicate, in this case:

> an unbound occurrence of y in the definition of f
> an unbound occurrence of x in the definition of g
> > which is defined in a

This output might be produced from the interaction

    Take input from where? *CHECK.LOB*

    Take input from where? *ERRONEOUS.LSO*
    ((y f) (x g a))

in case the file ERRONEOUS.LSO contained the text

    (let (f a)
     (f. (lambda (x) (cons x y)))
     (a. (let (g g) (g. (lambda (y) (cons x y))))))

Construction notes

CHECK.LSO compiles to give the closure in CHECK.CLS, and the code in CHECK.LOB is formed by linking

the closure	LOADS.LOB
the closure	CHECK.CLS
the argument count	1

Examples of applications

The remainder of the LispKit Lisp programs described in this manual are included, here and in the accompanying volume of source listings, not because they are components of the system, but to serve as examples both of the programming styles used by various members of the LispKit project and of the types of programs which have been written in LispKit Lisp over the past year.

A simulator for VLSI descriptions

This program is a simulator for a VLSI description language, μFP, which is based on Backus' FP language

A fuller description of the μFP language may be found in a D.Phil. thesis currently in preparation by Mary Sheeran at the Programming Research Group.

The values manipulated by a μFP program are thought of as being signals in the circuit being simulated, with each signal being represented by a time-sequence of values. The semantics of μFP are defined by a translation into FP, and this translation is used in the interpreter, followed by an interpretation of the resulting FP expression.

The design of the interpreter has intrinsic interest, since it consists of a skeleton which is largely independent of the interpreted language, and an evaluator and a library of definitions which together describe μFP. By using the skeleton of the interpreter, and substituting for the μFP evaluator and library, it should thus be possible with little effort to implement an interpreter for another language.

Included with the μFP interpreter is a file muFP_LIB.LSO of example definition commands which describe the construction of a full-adder circuit from primitive gates.

An outline of the FP language

The FP functions recognised by the interpreter are:

    id

which is the identity function;

    hd   tl

which take the head and tail, respectively, of lists;

    add   sub   mul   div   rem

each of which takes a two-list of numbers and returns the result of the corresponding Lisp arithmetic function applied to those numbers;

    eq

which compares the components of a two-list and returns 1 or 0 according as they are equal or not;

which returns 1 or 0 according as its argument is NIL or not;

zip

which performs a matrix transposition on an argument which is a list of lists;

appendl   appendr

which each take two-lists as arguments: appendl stands for 'append on the left', takes a value and a list, and conses the value on the front of the list; appendr takes a list and a value, and extends the list on the right by that value;

distl   distr

which take two-lists and 'distribute' one of the components through the other, so that distl which stands for 'distribute on the left' takes

(a (1 2 3 4))

into

((a 1) (a 2) (a 3) (a 4))

for example, and distr operates similarly, but on the right.

Additionally, there are a number of FP combining forms whose semantics are relevant to those of μFP. The combining forms are needed in FP because there is no direct way of defining higher order functions. The forms recognised by the interpreter are:

(select ⟨n⟩)

which is the function that selects the ⟨n⟩th component of its argument;

(compose ⟨exp1⟩ ⟨exp2⟩ ⟨exp3⟩ ⋯)

which is the composition of the component expressions;

(construct ⟨exp1⟩ ⟨exp2⟩ ⟨exp3⟩ ⋯)

whose value is the list of its component expressions;

(alpha ⟨exp⟩)

which is a function that expects a list as an argument and applies its component expression to each component of its argument, that is it'map's ⟨exp⟩ down its argument;

(slash ⟨exp⟩)

which is a function that expects a list as an argument and forms the continued application of its component expression through its argument, that is it 'reduce's ⟨exp⟩ down its argument;

```
(constant ⟨literal⟩)
```

which returns a constant function, whose value at any argument is
⟨literal⟩;

```
(if ⟨exp1⟩ ⟨exp2⟩ ⟨exp3⟩)
```

which takes an argument and applies the second and third component
expressions to that argument, returning the one result or the other
according as the result of applying the first component expression is 1 or 0.

An outline of the muFP language

Any FP expression is also a valid μFP expression, and represents the
pointwise application of the FP function to an input time-sequence of
values of the right type. Formally, any such function — which is called
'stateless' — is translated into a pointwise application by using the alpha
form in FP. Similarly, there are pointwise extensions of each of the FP
forms which define corresponding μFP forms. The detail of the translation
may be found in the function mufp in the text of muFP in the volume of
sources which accompanies this manual.

There is an important additional syntactic form, from which μFP derives
its name, and which is used to model circuit elements which have state.
Provided that ⟨f_exp⟩ is a function of type

```
input × state → output × state
```

and ⟨s_exp⟩ is of type state, then

```
(mu ⟨f_exp⟩ ⟨s_exp⟩)
```

is a function from input to output where the state is initially ⟨s_exp⟩ and
subsequently the state is fed back — with a delay of one time step — from
the result. An example is shown, in the interpreter session below, of the
simulation of a one stage shift register. The ⟨f_exp⟩ for this simulation is

```
(construct (select 2) (select 1))
```

which defines its output to be equal to its second argument (previous state),
and defines its next state to be a copy of its current input. The shift register is

```
(mu (construct (select 2) (select 1)) _!_ )
```

which has initial state, and so initial output,

```
_!_
```

— which we use to stand for 'not defined' — and then copies its input to its
output with a delay of one time-step.

Interpreter commands

The interpreter recognises a number of s-expression forms as commands which change its behaviour. The command

    (run ⟨exp⟩)

takes a μFP expression ⟨*exp*⟩, evaluates it, and applies it to an input sequence which is read from the input stream, terminating at the atom *end*. The output from the μFP expression is sent to the screen.

Names may be associated with expressions by the command

    (def ⟨name⟩ ⟨expression⟩)

which allows the ⟨*name*⟩ to stand for the ⟨*expression*⟩ and gives the capability for making recursive definitions. These definitions may then be edited by typing the command

    (edit ⟨name⟩)

which invokes the editor E, described elsewhere in this manual, to edit the text of the definition of ⟨*name*⟩. When the editor is left by typing *end* the interpreter expects another command.

The command

    vars

displays a list of all defined names, and

    dump

outputs a sequence of *def* commands which can be stored in a file and re-input to the interpreter to redefine these names. Any individual definition may be seen in a more readable form by using the command

    (show ⟨name⟩)

which pretty-prints the definition of ⟨*name*⟩. A definition is revoked by the

    (cancel ⟨name⟩)

command which removes the definition of ⟨*name*⟩. Finally, the command

    end

ends the execution of the interpreter.

An example muFP interpreter session

Execution of the interpreter begins

    Take input from where? *muFP.LOB*
    muFP Interpreter

    ⟩
    Take input from where? *CONSOLE:*

First, we will introduce the one stage shift register example referred to above; we begin by defining the transition function:

```
(def two_one (construct (select 2) (select 1)))
 ) (run two_one)
(a b)
( b a ) (c d)
( d c ) (1 2 3)
( 2 1 ) end
```

then a one stage shift register with an initially undefined state:

```
 ) (def shift1 (mu two_one _|_))
 ) (run shift1)
0
__ 1
0 2
1 3
2 end
```

and now a two stage shift register composed from two one stage registers:

```
 ) (run (compose shift1 shift1))
0
_|_ 1
_|_ 2
0 3
1 4
2 5
3 end
```

The next example shows a somewhat larger μFP expression with differing amounts of delay in each output:

```
 ) (def rtriang
     (compose
       (slash
         (compose
           appendl
           (construct
             (select 1)
             (compose shift1 (select 2)) )))
         appendr
         (construct id (constant NIL)) ))
 ) (run rtriang)
(0 0 0)
( 0 . _|_ ) (0 0 0)
( 0 0 . _|_ ) (1 1 1)
( 1 0 0 . _'_ ) (2 2 2)
( 2 1 0 ) (3 3 3)
( 3 2 1 ) (0 0 0)
( 0 3 2 ) (0 0 0)
( 0 0 3 ) end
```

Finally, an example of how to use the editor to correct mistakes in definitions:

```
> (def addup (slash sub))
> (edit addup)
p
  ( slash sub ) (e sub add) end
  > (show addup)
( def addup ( slash add ) )
  > (run addup)
(0 0 0 0)
0 (1 1 1 1)
4 (1 2 3 4)
10 end
  > end
Exit muFP interpreter
```

Construction notes

The text in mu FP.LSO uses the following libraries

```
E_CONTROL.LIB
E_FUNCTION.LIB
E_MATCH.LIB
E_MISC.LIB
S_EXPRESSION.LIB
ASSOCIATION.LIB
STANDARD.LIB
SECD_CODE.LIB
```

and the composite text compiles to give the code in muFP.LOB.

A simple semantic network database

This note reports on the development of a simple program for storing and interrogating information stored as a semantic network. In such a network we have labelled nodes and labelled, directed arcs. We use each
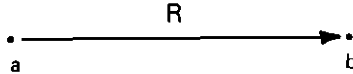
arc to record a relationship aRb. For example, the facts that John and Mary love each other and that both love Logic are recorded in the following net, as are the facts that John is a Male, and Mary is a Female, and that they are both Programmers:
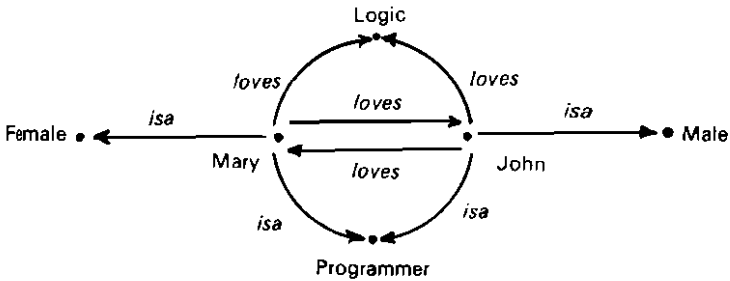
Methods of recording complex facts in semantic networks are well known

We shall design and build a LispKit Lisp program which allows us to manipulate a semantic network represented as an s-expression. The important point to note from the example is that we shall be required to deal with sets of nodes, for example, all those things which John loves, that is

{Logic,Mary},

and so that shall be our starting point.

A calculator for sets

Consider the standard LispKit Lisp idiom for an interactive program.

```
(letrec (lambda (kb) (map eval (untilend kb)))
   (eval. (lambda (x) (cons x x))) )
```

In this, every item the user types, until he types *'end'*, is submitted to eval and the result is printed on the screen. Our example has a rather uninspiring eval which, given x, simply constructs the pair (x.x). Nevertheless, this program has the form of a calulator, responding to every typed item by printing a pair made from two copies of that item.

To make the calculator do something useful let us define a simple language of set valued expressions using the usual set operations

```
⟨expr⟩   ::=   ⟨set⟩
         |     (union ⟨expr⟩ ⟨expr⟩)
         |     (inter ⟨expr⟩ ⟨expr⟩)
         |     (diff ⟨expr⟩ ⟨expr⟩)
⟨set⟩    ::=   ⟨atom list⟩
```

A constant set is denoted by a list of atoms without repetition. For example, each of

```
(a b c)
(1 2 3 4 5 6)
(1 a 3 c −1)
```

is a set. The user of our set calculator may type any ⟨expr⟩ and the calculator will evaluate it. For example, on his typing

```
(diff (union (a b c) (c d e)) (a d f))
```

the calculator should respond

```
(b c e)
```

Consider how we might recognise that the ⟨expr⟩ has union as its operator. That is, that it is of the form

```
(union ⟨expr⟩ ⟨expr⟩).
```

The following predicate would serve:

```
isunion = (lambda (e) (and (not (atom e))
                           (eq (head e) (quote union))))
```

Similarly, we have predicates for recognising the other well-formed ⟨expr⟩s

```
isinter = (lambda (e) (and (not (atom e))
                            (eq (head e) (quote inter))))
isdiff = (lambda (e) (and (not (atom e))
                          (eq (head e) (quote diff))))
```

If the ⟨expr⟩ does not satisfy any one of these predicates, then we check to

see whether it is a list of atoms, a constant set value. Hence

```
isatomlist = (lambda (e)
              (or (eq e (quote NIL))
                 (and (not (atom e))
                      (and (atom (head e)) (isatomlist (tail e)))))))
```

is deployed.

Dismantling the ⟨expr⟩ once we have recognised that it has an operator and two operands can be done safely using functions **arg1** and **arg2** which select the first and second operands, if available, and return the empty set — that is, **NIL** — if not.

```
arg1 = (lambda (e)
              (if (leq (quote 2) (length e))
                      (head (tail e)) (quote NIL)))
arg2 = (lambda (e)
              (if (leq (quote 3) (length e))
                      (head (tail (tail e))) (quote NIL)))
```

Thus with this scaffolding we erect a new eval function which accepts utterances in our language of set valued ⟨expr⟩s and which computes the set which each utterance denotes.

```
eval = (lambda (e)
              (if (isunion e)
                   (union (eval (arg1 e)) (eval (arg2 e)))
              (if (isinter e)
                   (intersection (eval (arg1 e)) (eval arg2 e)))
              (if (isdiff e)
                   (difference (eval (arg1 e)) (eval (arg2 e)))
              (if (isatomlist e)
                      e
                 emptyset )))))
```

Slotted into our calculator program in place of our trivial eval, we have already a not uninteresting application.

Semantic network representation

We choose to represent a network using association lists as in

'Lisp',   P. H. Winston and B. K. P. Horn,
Addison Wesley.   Reading, Massachusetts, 1981

An association list can be thought of as a finite function mapping atoms to values. It is represented as a list of pairs, each pair consisting of an atom and a value. The standard operations are

```
defined :   atom  ×  association list   →   Boolean
associate : atom  ×  association list   →   value
```

which respectively tell us whether there is a binding of a particular atom in a particular association list, and return the corresponding value when there is

one. Let us refer to the set of atoms which are defined in a particular association list as the domain and the set of values onto which they map as the range of the association list.

The network will be represented by an association list whose domain is the set of nodes, and whose range is a set of association lists. These embedded association lists will have sets of relation names (arc labels) as their domains and sets of nodes as their ranges. For example, the earlier example has the representation

```
((John.((loves.(Logic Mary))
        (isa.(Male Programmer))))
 (Mary.((loves.(John Logic))
        (isa.(Female Programmer)) )) )
```

Hence we could discover, for example, those things which John loves by evaluating

```
(associate (quote loves) (associate (quote John) db))
```

where **db** is the above network representation.

Now consider how we might give our set calculator the ability to compute using sets derived from such a semantic network. Suppose we add the following syntax to the language of our calculator

```
⟨expr⟩  ::=  (im ⟨expr⟩ ⟨attr⟩)
⟨attr⟩  ::=  ⟨atom⟩
```

Here we intend to allow utterances of the form

(im (John) loves)

to denote those things which John loves, and

(im (John Mary) isa)

to denote those things which either John is or Mary is, that is

(Male Female Programmer).

So the utterance

(im ⟨expr⟩ ⟨attr⟩)

denotes the image of the set ⟨*expr*⟩ under the relation ⟨*attr*⟩. From hereon in we shall refer to the relation R in **aRb** as an attribute of the object **a**.

Power comes when we realise that

(inter (im (John) loves) (im (Mary) loves))

denotes those things which both John and Mary love, that is (Logic), and

(diff (im (John) loves) (im (Mary) loves))

those things which John loves, but that Mary does not, that is (Mary)!

So how are we to extend our calculator? First we note that **eval** has to have a second argument, which is the network, and we christen this **db**, for database.

```
eval = (lambda (e db)
         (if (isunion e)
             (union (eval (arg1 e)) (eval (arg2 e)))
         (if (isinter e)
             (intersection (eval (arg1 e)) (eval arg2 e)))
         (if (isdiff e)
             (difference (eval (arg1 e)) (eval (arg2 e)))
         (if (isim e)
             (imageset (eval (arg1 e) db) (mkatom (arg2 e)) db)
         (if (isatomlist e)
             e
           emptyset )))))
```

Here we have relied on the obvious

```
isim = (lambda (e) (and (not (atom e)) (eq (head e) (quote im))))
```

and the possibly unnecessary

```
mkatom = (lambda (x) (if (atom x) x (quote NIL)))
```

to analyse the $\langle expr \rangle$ and in case it is of the form

(im $\langle expr \rangle$ $\langle attr \rangle$)

to submit it to the function imageset to compute the denoted value.

Consider what imageset must do. Given a set of nodes and an attribute it must compute

$$\bigcup_{n \in nodeset} image(n, attr, db)$$

That is, for each n in the given node set it must compute the union of the set of nodes which are in the image of that node with respect to the given attribute. Such an expression has a direct, if clumsy representation in LispKit Lisp.

```
imageset = (lambda (nodeset attr db)
              (reduce
                union
                (map (lambda (n) (image n attr db)) nodeset)
                emptyset))
```

By mapping the function

$\lambda n.image(n, attr, db)$

over the node set we compute

{ image(n, attr, db)   n $\in$ nodeset }

and by reducing this set of node sets using set union we flatten it into the required node set.

The work of interrogating the semantic net has been left to the function image, which has a direct rendition using a double application of associate as described above.

```
image = (lambda (node attr db)
           (if (defined node db)
                  (let (if (defined attr a) (associate attr a) emptyset)
                       (a.(associate node db))
                  emptyset))
```

Note that if we ask about a node outside the domain of db, or an attribute outside the domain of an embedded association list, then we compute the empty set as default.

So our new calculator is complete. It allows a limited form of interrogation of a semantic net. The database can be constructed elsewhere and given to the calculator in the conventional way, as the first object on the input stream. Thus the driving function becomes

```
(lambda (kb)
   (map (lambda (c) (eval c (head kb))) (untilend (tail kb))))
```

as we have come to expect with this type of interactive LispKit program.

A small extension

Now we extend the language of the calculator to allow the utterance

(inv ⟨attr⟩ ⟨expr⟩)

which denotes the set of nodes which have the given attribute in the set ⟨expr⟩. For example

(inv loves (Logic))

denotes those things which love *Logic*, that is

(John Mary).

We call it the inverse image of ⟨expr⟩ under ⟨attr⟩. This is surprisingly easy to compute, and powerful. Are there any Programmers who love Logic?

(inter (inv loves (Logic)) (inv isa (Programmer)))?

Yes there are. Both John and Mary in fact.

The standard function domain, applied to an association list, returns the set of atoms defined to have values by that association list. Given this we can define

```
invimage(nodeset,attr,db)  =
               { n∈domain(db) . image(n,attr,db)∩nodeset ≠ ∅ }
```

This reads as follows: the inverse image of nodeset under the given attr is the set of all those nodes n such that the image of n under attr has some element in common with nodeset. This may be rendered into LispKit Lisp as follows:

```
invimage =
  (lambda (nodeset attr db)
    (filter
      (lambda (n)
          (not (eq (intersection (image n attr db) nodeset) emptyset)))
        (domain db) ))
```

The extensions required to eval are obvious, so we do not list them here.
The final version of the program, developed below, is listed in the
companion volume to this, and may be inspected for these extensions.

Updating the database

We consider how new components may be added to the semantic network.
Let us postulate an outer layer of command structure for our calculator.

```
⟨command⟩  ::=  (add ⟨expr⟩ ⟨attr⟩ ⟨expr⟩)
             |    db
             |    ⟨expr⟩
```

Here we have introduced three commands. The command

> *(add ⟨expr1⟩ ⟨attr⟩ ⟨expr2⟩)*

adds to the semantic net all the associations between members of ⟨**expr1**⟩
and ⟨**expr2**⟩ under attribute ⟨**attr**⟩. Thus

> *(add (John Mary) dislikes (Fortran Messiaen))*

adds four facts to the database. The command

> *db*

causes the entire database to be printed to the output stream, the
conventional method of saving an updated state. Finally, the command
⟨**expr**⟩ evaluates the denoted set, as before, and prints it. The driving
program then becomes

```
(lambda (kb) (execute (untilend (tail kb)) (head kb)))
```

and the outer loop is implemented by the function execute.

```
execute = (lambda (c db)
            (if (eq c (quote NIL))
                (quote NIL)
            (if (isadd (head c))
                (execute
                    (tail c)
                    (addset db
                            (eval (arg1 (head c)) db)
                            (mkatom (arg2 (head c)))
                            (eval (arg3 (head c)) db)))
            (if (isdb (head c))
                (cons db (execute (tail c) db))
                (cons (eval (head c) db) (execute (tail c) db)))))))
```

Thus we see that there are two categories of command. The add command computes a new database by calling addset with its evaluated arguments, but produces no response, while the other two commands produce responses but proceed with the execution on an unaltered database.

So now to addset. Let us delegate the responsibility for updating a single node to a function addel (for 'add element'). This function will be given a node, an attribute, a node set and the database. It will compute the database which records that this node has this attribute in addition to all of its pre-existing attributes, whether or not it previously had this attribute; also, that this node is related by this attribute to each node of the node set, in addition to any pre-existing relationships.

db and *addel(0, a, {3, 4, 5}, db)*
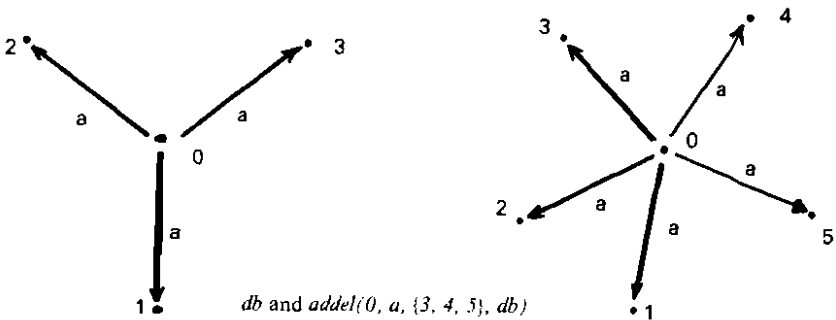
The composition of

    addel(node,attr,nodeset,db)

from db is straightforward but we must be a little cautious to deal properly with nodes and attributes not yet recorded in the database. The standard function

    update · association list × atom × value → association list

is used. This computes the association list with the pairing

    (atom.value)

either replacing any other pairing for atom, or as a new element if there is no pairing for atom. Thus we have

```
addel = (lambda (node attr nodeset db)
           (let
               (let (update db node (update a attr (union nodeset s)))
                  (s.(if (defined attr a)
                         (associate attr a)
                         emptyset )) )
               (a.(if (defined node db)
                      (associate node db)
                      (quote NIL) )) ) )
```

Note the definition of a, which protects us against a node which does not yet appear, and of s, which similarly protects us against a missing attribute. Given that a is the association list paired with node in db, and that s is the set paired with attr in a, then

```
(update db node (update a attr (union nodeset s)))
```

computes the database with the additional associations which we require.

So we come finally to the function addset. This is defined from addel by using reduce, as follows

```
addset = (lambda (db ns1 attr ns2)
              (reduce (lambda (n db) (addel n attr ns2 db)) ns1 db))
```

The consequence of this is that each n in the set ns1 is added to db, associated with the set ns2. That is, we compute

```
(addel n1 attr ns2 (addel n2 attr ns2 ⋯ (addel nk attr ns2 db)⋯))
      where ns1 = (n1 n2 ⋯ nk)
```

and so each new association is recorded.

Sample session

The following session is started from the semantic network stored in the file SEMNET_LIB.LSO. which is listed in the companion volume to this.

```
Take input from where? SEMNET.LOB

Take input from where? SEMNET_LIB.LSO

Take input from where? CONSOLE:
(im (Mary) loves)
( Whiskey Logic John )
(inv loves (Logic))
( John Mary )
(union (inv loves (John)) (inv loves (Mary)))
( Mary John )
(inv isa (Giving))
( Giving1 Giving2 Giving3 )
(im (Giving1) giver)
( John )
(im (Giving1) givee)
( Mary )
(im (Giving1) given)
( Book )
(im (inv isa (Giving)) given)
( Book Flowers Kiss )
(im (inter (inv giver (John)) (inv givee (Mary))) given)
( Book Flowers )
(add (John Mary) dislikes (Fortran Messiaen))
(inv dislikes (Messiaen))
( John Mary )
```

(im (John) dislikes)
(Fortran Messiaen)
(add (inv loves (Logic)) loves (Poetry))
(im (John) loves)
(Logic Mary Poetry)
(inv loves (Poetry))
(John Mary)
(dill (im (John) loves) (im (Mary) loves))
(Mary)
(add (Giving) isa (Action))
(add (Giving) includes (inv isa (Giving)))
(im (Giving) includes)
(Giving1 Giving2 Giving3)
end

Construction notes

SEMNET.LSO requires the libraries

SET.LIB
ASSOCIATION.LIB
STANDARD.LIB

and the composite text compiles to give the code in SEMNET.LOB.

An interpreter for a logic language

The program LOGIC is an interpreter for a logic language, essentially a dialect of PROLOG free of side effects. The interpreter is closely based on the LispKit Lisp interpreter, and the reader is assumed to have read the manual for that program, INTERP. The interpreted language is based on LogLisp, which is described in

LOGLISP: Motivation, Design and Implementation,
J. A. Robinson and E. E. Sibert
in Logic Programming, ed K. L. Clark and S.-A. Tarnlund
Academic Press, London, 1982

It involves the manipulation of a database of assertions, both by commands which change the database, and by expression forms which interrogate the database. These are all in addition to the LispKit Lisp interpreter commands, and the LispKit Lisp expression forms, as described in the section of the manual on the program INTERP.

The database may be thought of as asserting some relationships between LispKit data objects. These relationships are represented by s-expression forms, for example

    (deserts are dry)

Any s-expression form either does or does not represent a truth, in the context of a particular database. A form is true with respect to a database either if the form was asserted whilst creating the database, or if the form is deducible from the assertions that have been made.

Commands for making assertions

Initially, there are no relationships which are asserted in the database. The command

    (fact ⟨conclusion⟩)

adds to the database the assertion that the s-expression for ⟨*conclusion*⟩ represents a true relationship. Thus, for example, the command

    (fact (deserts are dry))

would assert the truth of the statement that

    (deserts are dry)

More generally, the command

    (fact ⟨conclusion⟩ . ⟨hypothesis list⟩)

asserts that whenever each of the statements in the ⟨*hypothesis list*⟩ is true, then the ⟨*conclusion*⟩ is also true.

Large numbers of systematically related facts may be asserted simultaneously by the command

```
(forall ⟨variable list⟩ ⟨conclusion⟩ . ⟨hypothesis list⟩ )
```

The ⟨*variable list*⟩ is just a list of atoms which represent dummy variables in the command. A *forall* command asserts that if each hypothesis in the ⟨*hypothesis list*⟩ is true under any substitution of expressions for the variables in the ⟨*variable list*⟩, then the conclusion is also true under that substitution. Thus the command

```
(forall (x) (x are unpleasant) (x are dry))
```

(read 'any x are unpleasant if those x are dry') would add to the database in our example the assertion that

```
(deserts are unpleasant)
```

The *fact* command is just an abbreviation for

```
(lorall NIL ⟨conclusion⟩ . ⟨hypothesis list⟩)
```

Finally, there is a command

```
new
```

which retracts each of the assertions previously made, allowing the user to start afresh with an empty database.

Additional expressions

There is an expression which allows queries to be made of the database, namely

```
(all ⟨variable list⟩ . ⟨condition list⟩)
```

This is treated as any other expression, as though it were a LispKit Lisp expression. Its value is a list of all the substitutions for the variables in the ⟨*variable list*⟩ under which every one of the conditions in the ⟨*condition list*⟩ is true. Thus, in the context of the desert database which was constructed above, the value of

```
(all (x) (deserts are x))
```

would be a two element list

```
( ( dry ) ( unpleasant ) )
```

The value of an *all* expression may be empty, indicating that there are no substitutions which make all the conditions necessary concomitants of the assertions in the database; the value may be infinite; it may contain repetitions; it may even contain variables.

If the value of an *all* expression contains logical variables, then they will be printed in the form

(⟨variable name⟩ : ⟨subscript⟩)

In this case, the conditions of the enquiry are true no matter what expression is substituted for the logical variables. Notice that although these subscripted variables are printed as lists, the LispKit Lisp predicate

(atom ···)

is true of of all logical variables.

The other new expression form which is recognised by the interpreter allows assertions to be added temporarily to the database, and then removed after an enquiry is complete. The value of

(logic ⟨body⟩ . ⟨assertion list⟩)

is the value of the ⟨*body*⟩ with respect to a database in which each of the assertions of the ⟨*assertion list*⟩ has been added to the database. The evaluation of this expression does not, of course, affect the database. For an example of the use of the logic expression, see the restore file LOGIC_LIB.LSO which contains an expression that defines the palindromic lists.

An example session using the interpreter

```
Take input from where? LOGIC.LOB
Logic LispKit interpreter: type end to finish
 ⟩
Take input from where? CONSOLE:
(fact (John likes Mary))
asserted
 ⟩ (all (x y) (x likes y))
( ( John Mary ) )
 ⟩ (forall (x) (Fred tolerates x) (x likes Mary))
asserted
 ⟩ (forall (x) (Jean.x) (Fred.x))
asserted
 ⟩ (forall (x y) (x tolerates y) (x likes y))
asserted
 ⟩ (all (x y) (x tolerates y))
( ( John Mary ) ( Fred John ) ( Jean John ) )
 ⟩ new
new database
 ⟩ (all (x y) (x likes y))
NIL
 ⟩ end
Exit logic interpreter
```

Construction notes
The source in LOGIC.LSO is complete in itself, and compiles to give the object in LOGIC.LOB.

A script-driven adviser

EXPERT is an interpreter for a language intended to control conversations. The primitives of the language provide for sentences to be output by the program, for responses to be elicited to questions, and for salient parts of the history of a conversation to be recorded. An example is given, in the companion volume, of a script for governing a conversation which gives guidance on a repair task.

The program expects an s-expression at the head of its input stream which is a script. Nodes in the script describe the state of an automaton which proceeds from state to state, writing statements and questions to its output stream, and reading replies from the remaining input. The course taken through the script may be determined by the replies, by 'hard wired' decisions in the script, or by the values of variables which are changed as determined by the script.

A script is an association list, in which each binding represents a state, which has a name that is bound to a list of instructions. There is one particular state, named init, in which execution begins. The machine then obeys each of the instructions bound to the name of its state until the state is changed, or there are no instructions left. Execution terminates when the machine enters the state named end. The

```
(say . ⟨exp⟩)
```

instruction writes the s-expression ⟨exp⟩ to the output stream.

```
(ask ⟨var⟩ . ⟨exp⟩)
```

writes ⟨exp⟩ and then expects a reply from the input. The reply is a single s-expression which is then bound to the atomic name ⟨var⟩ in an association list of variables which the program maintains; at the same time, the value is placed in an accumulator which is referred to by conditional instructions. Variables may also be bound to values by the explicit assignment

```
(note ⟨var⟩ ⟨exp⟩)
```

which binds the value ⟨exp⟩ to the variable ⟨var⟩.

```
(getval ⟨var⟩)
```

loads the accumulator with the value bound to the variable named ⟨var⟩.

```
(enq ⟨var⟩ . ⟨exp⟩)
```

combines the functions of ask and getval since it loads the accumulator from the variable ⟨var⟩ provided that it is already defined, but if it is not defined, then an ask is performed first. The current state may be changed by a

```
(goto ⟨state⟩)
```

instruction which causes execution to resume at the first instruction of the state ⟨state⟩, or by

    (if ⟨val⟩ ⟨state⟩)

which changes the state only if the accumulator contains the specified value, or

    (ifn ⟨value⟩ ⟨exp⟩)

which performs the change of state only if the accumulator does not contain the specified value.

   In any expression, the symbol * may be used to stand for the value of the accumulator. In particular, the instruction

    (goto *)

can be used to implement 'return from subroutine' jumps.

Example of a conversation

    Take input from where? *EXPERT.LOB*

    Take input from where? *EXPERT_LIB.LSO*
    Kelloggs Delegging Machine (Model A) service routine

     Is the button pressable?
    Take input from where? *CONSOLE:*
                        yes
    Do all the legs fall off? *no*
    Do any of the legs fall off? *no*
    Are there any legs? *yes*
    Inspect the non – departed legs:
     Have they been nailed on? *no*
    Is there any sign of glue? *yes*
    Soak legs in paradi – chloro – phenyl – pentanoic acid
     Press the button again
     Do all the legs fall off? *no*
    Do any of the legs fall off? *no*
    Fit a new button assembly (part no #765/wy/35454y/z2)
     Press the button again
     Do all the legs fall off? *yes*
    Is the machine clean and shiny? *no*
    Sponge the machine with warm soapy water, taking care not to wet
     the expeditionary telephone emulator or the cold air intake conduit
     Is the machine clean and shiny? *yes*
    Your Kelloggs Delegging Machine (Model A) is in perfect working order

Construction notes
EXPERT.LSO requires the libraries

    STANDARD.LIB
    SECD_CODE.LIB

and the composite text compiles to give the code in **EXPERT.LOB**. Since this program requires a script as its first input, you may choose to link this code with a particular (uncompiled) script, such as that in **EXPERT_LIB.LSO** to make a program that is specific to one type of conversation.

nFib - A speed benchmark

The function nFib is included in this manual, not for any intrinsic interest, but because it is used as a speed benchmark for each new implementation of the LispKit system. The function, defined by

$$nFib(0) = 1$$
$$nFib(1) = 1$$
$$nFib(n+2) = nFib(n+1) + nFib(n) + 1$$

is used to calculate the number of function invocations required to calculate its own result. Thus, for example

$$nFib(15) = 1973$$

and the algorithm used to calculate this value executes one thousand nine hundred and seventy three function calls. The closed form of the definition of nFib is

$$nFib(n) = \frac{5+\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^n + \frac{5-\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^n - 1$$

$$= 2\,Fib(n) - 1$$

(Fib(n) being the nth Fibonacci number) showing that nFib(n) makes a number of function calls exponential in the argument n. Each of the timings reported in the final sections was made with a call of nFib on an argument chosen to make the call last for about half a minute.

Construction notes

The text in NFIB.LSO is complete in itself and compiles to give the closure NFIB.CLS. The code in NFIB.LOB is formed by linking

the closure	**LOADS.LOB**
the closure	**NFIB.CLS**
the argument count	1

Examples of the use of infinite objects

One of the most striking uses of lazy evaluation is in the description of computations which yield infinite objects. A LispKit Lisp expression may have a value which is infinitely large; for example, the input stream is a list with infinitely many components. Such an expression describes the infinite computation which would, as it was executed, generate more and more of the value of the expression. The order of execution of the computation imposed by lazy evaluation ensures that the only work done in a computation is that which is essential to the result. Thus, if the result of a program depends only on the result of finitely much of an infinite computation, that program will still terminate. If, however, the whole of an infinite object is to be calculated, for example, if a LispKit Lisp program defines an infinite output stream, then the output will be produced piecemeal, as the computation progresses.

Four examples of LispKit Lisp expressions whose values are infinite lists (streams) are listed in the companion volume to this. Each is a source which may be compiled to produce a .CLS file which can then be executed with the LOADK loader.

The first, INTEGERS, evaluates to a list which consists of the non-negative numbers (natural numbers) and is an example of a program that will produce arbitrarily much output, since it runs in a bounded space. Of course, the number range will eventually be exceeded, and overflow will occur, but the program will continue counting up until interrupted.

The second expression, PRIMES, describes the list of prime integers, using the Sieve of Eratosthenes. Since this method requires all smaller primes to be available for the calculation of the next prime, the program gradually accumulates state, and an attempt to evaluate the whole of the list of primes will eventually lead to the cell store being filled.

The expression ROUND is an example quoted by Gilles Kahn to demonstrate the power of languages which process infinite objects. Its value is the list consisting of all the round numbers, that is those whose only prime factors are two, three, or five. The list is generated in ascending order by merging three other lists, each of them also infinite, each defined in terms of the list of round numbers itself. The algorithm is presented in

A Discipline of Programming, E. W. Dijkstra
Prentice-Hall. New Jersey 1976, 0-13-215871-X

where the problem is attributed to R. W. Hamming.

Finally, the expression EDIGITS evaluates to the list of the digits of e, the base of natural logarithms. This program is a slight modification of a program by David Turner (who attributes the algorithm to

E. W. Dijkstra) which is described in

Recursion equations as a programming language, D. A. Turner
in *Functional Programming and its Applications,*
ed J. Darlington, P. Henderson, D. A. Turner
Cambridge University Press, 1982, 0-521-24503-6

and calculates the sequence of digits by a process of translating the continued sum

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

into a decimal expansion.

Construction notes

Each of the four .LSO files contains a complete expression, and may be compiled to yield the corresponding .CLS file, thus

```
⟨ctrl Y⟩⟨return⟩
Send output to where? EDIGITS.CLS
⟨ctrl Z⟩

Take input from where? LC.LOB

Take input from where? EDIGITS.LSO

Take input from where? CONSOLE:
⟨ctrl Y⟩⟨return⟩
Send output to where? CONSOLE:
```

The following type of interaction may be used to evaluate one of these expressions

```
Take input from where? LOADK.LOB

Take input from where? ROUND.CLS
1 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 ···
```

or, of course, the codes may be linked to form complete programs, ROUND.LOB for example consisting of

the closure for LOADK	LOADK.LOB
the closure for the expression	ROUND.CLS

Libraries of useful standard functions

A number of expression libraries are listed in the volume of sources which accompanies this manual. By convention, these are given file names with the extension

.LIB

Of these, some are components of specific programs, and are named accordingly, for example

E_FUNCTION.LIB, E_MATCH.LIB and E_MISC.LIB

contain portions of the list structure editor E. Other libraries are more general, and contain suites of related functions which are intended to be more generally useful.

A library is an association list, of the same form as the definition part of a letrec-expression. Used with the library manager, LIBMAN, it extends the LispKit Lisp language by adding definitions of extra standard functions and values. Programs may be written assuming the definitions of such functions as append and equal, and these definitions added automatically by the library manager, to produce a complete LispKit Lisp program.

The library ASSOCIATION

This library contains functions for manipulation of association lists, such as libraries. An association list is a list of pairs: the head of each pair is an atom, called the 'name' of the pair, and the tail of each pair is a value which is said to be 'bound' to that name.

(domain a)	is the list of names bound in the association list a
(defined e a)	is true if some value is bound to the name e in the association list a
(associate e a)	is the value bound to the name e in the association list a
(bind e d a)	is an association list containing a binding of d to the name e, in addition to all the bindings of the association list a
(unbind e a)	is an association list containing the bindings of the list a, except that the name e is not bound
(update a e d)	is an association list containing the bindings of the list a, except that d is bound to the name e

The library SECD_CODE

The definitions in this library are either hand written code objects — which are mainly sequences of opcodes which could not be produced by compilation of LispKit Lisp source — or are functions which construct LispKit Lisp code objects. They provide a small interface between the LispKit programmer and the virtual machine. It is to be expected that the implementation of most of these definitions could change in future implementations of the LispKit system, but that their specifications would remain essentially unchanged. The implementor's guide describes the relevant machine concepts in more detail.

The functions in this library fall into three distinct classes according to the type of programs in which they can safely appear.

Any program may reasonably use the function flexible which takes a function of one argument, and returns a function of an unspecified number of arguments, so that

    ((flexible f) a b c d ⋯) = (f (list a b c d ⋯))

Similarly, the function chr behaves as a perfectly ordinary LispKit Lisp function, such that

    (chr n)

is an atom consisting of the single character with ASCII code n.

The following functions may be used — usually in 'systems programs' — with the exercise of care in controlling the order of execution. They manipulate data structures which have specific meanings to the virtual machine; for example, code which is executed by the machine must be fully evaluated before execution begins, and the function make_closure takes an expression whose value is code, and returns a fully evaluated copy of the code.

(apply_code f l)	applies the evaluated closure f to the evaluated argument list l
(make_closure f)	evaluates the closure f
(make_arglist l)	evaluates thae argument list l
(inspect_code f)	used by make_closure
(inspect_env e)	used by make_closure
(inspect_arglist l)	used by make_closure and by make_arglist
(strict_cons h t)	returns the same value as (cons h t) but also evaluates both h and t before doing so
(sequence a b)	returns the value of b having first evaluated a
(finite e)	is true provided that e is a finite data structure, and has the side effect of forcing the evaluation of the whole of e

The following are not real functions, and should appear only in bootstrap loaders and in the run time system.

(run_and_halt x)	takes an application x and evaluates it, and never returns
(print_item x)	sends a representation of x to the output and returns x
(read_item)	reads a value from the input, and returns that value

.

The library SET

This library contains a number of functions for manipulating lists which are to be treated as sets (in the mathematical sense) of atoms. The membership test for such sets is the function member, defined in STANDARD.LIB.

emptyset	is the set with no elements
(singleton e)	is the set containing only e
(addelement e l)	is the set containing e, and all the elements of l
(remelement e l)	is the set containing all of the elements of l, excepting e
(union a b)	is the set containing every element of either of a or b
(intersection a b)	is the set containing every element common to a and b
(difference a b)	is the set containing all elements of a excepting those which are also elements of b

The library S_EXPRESSION

This library contains a number of functions relating to the representation and formatting of data structures.

    (dump n f)

is a structure which is the same as f except that all sub-structures more than n levels away from the root have been removed and replaced by atoms containing a single asterisk. This function is used to give an impression of the overall structure of the expression.

    (flatten s c)

is a list which consists of a flat representation of the structure s, concatenated with the list c. In particular

    (flatten s (quote NIL))

is the flat representation of s, that is, it is a list of the atoms of s separated by atoms containing the parenthesis symbols, and full stops, arranged so that this sequence of atoms is the representation of s as an s-expression.

    (pretty s)

is a structure containing each of the atoms of s, in a similar disposition, but interleaved with space and newline atoms in such a way that the printed representation of the result is layed out and indented so as to reflect the nesting structure of s.

The library SORT

This library presently contains only one function definition, that of the function quicksort. If less is a function for which

> (less a b)

is true of items a and b of some type, precisely when a is 'less' than b, then

> (quicksort less)

is a function which sorts a list of items of that type into ascending order by Hoare's quicksort algorithm. For example,

> (quicksort (lambda (a b) (if (eq a b) (quote F) (leq a b))))

is a function that expects a list of numbers as its argument, and returns a list of the same numbers sorted into ascending order.

The library STANDARD

The definitions in this library are those which might be expected to be required by almost any LispKit Lisp program. They are mostly functions which will be well known to any programmer with experience of some list oriented functional language.

List processing functions

(append e1 e2)	is the list obtained by concatenation of its two arguments
(member e l)	is T if the atom e is present in the list l
(equal e1 e2)	tests the equality of two non-function data structures
(null e)	is T if e is NIL, and is F otherwise
(length l)	is the number of components in the list l
(first n l)	returns a prefix of the first n components of the list l, or the whole list if shorter
(list e1 ⋯ en)	returns a list of its arguments
(transpose m)	is the matrix transposition of a list of lists m

Logical operators

(not c)	is T if c is F, and is F if c is T
(or c1 c2)	is T if either of the arguments is T, and is F if both are F
(and c1 c2)	is T if both of the arguments are T, and is F if either is F
(unless c1 c2)	is an abbreviation for (and (not c1) c2)

Stream operations

(until e l)	is the list of components of the stream l which precede the first occurrence of the atom e
(untilend l)	abbreviates (until (quote end) l)
(after e l)	is the stream of components of l which follow the first occurrence of the atom e
(afterend l)	abbreviates (after (quote end) l)

Commonly used higher order functions

(map f l)	is the list whose components are obtained from those of l by application of f
(reduce f l z)	is the continued application of f over the list l with zero z, that is (f (head l) (f (head (tail l)) (f ⋯ z)⋯))
(filter p l)	is the list of those components c of l for which (p c) is T

(close r l)	is the first value x in the sequence
	l, (r l), (r (r l)), ⋯
	for which
	(equal x (r x))

The remaining definitions conceal details of the implementation of the LispKit system which should not be written into every LispKit Lisp program that the user writes.

(number x)	is T if x is a numeric atom, and F otherwise
(load_code c)	is the value represented by the code object c. See the section on separate compilation, and the implementor's guide.
(apply f l)	applies the function f to the argument list l
newline	an atom which prints as a line break
space	an atom which prints as a blank space

The library TUPLE

This library contains definitions of a number of functions which are convenient when accessing components of a list.

(el n t)	is the nth component of the list t
(1 t)	is (el (quote 1) t), the first component of the list t
(2 t)	is (el (quote 2) t). the second component of the list t

and so on up to

(6 t)	which is (el (quote 6) t), the sixth element of the list t

The virtual machine

The Pascal source listing of the virtual machine describes the interface between the LispKit system and its host machine. This program is the only part of the whole system which is not source and object code portable from implementation to implementation. The SECD machine is modelled on Landin's SECD interpreter for applicative forms, described in

The mechanical evaluation of expressions,   *P. J. Landin*
in The Computer Journal, 308-320, January 1964

A fuller description of the virtual machine may be found in the book, which should be read along with those sections of the implementor's guide which describe the instructions which were added in order to implement lazy evaluation.

The source listed in the companion volume to this manual is in standard Pascal (in as far as such a standard exists) and is complete but for the absence of the definitions of a handful of routines at the head of the program. The routines are those which describe the interface to the underlying filing system. Example codings of these routine are given for various dialects of Pascal, and for different operating systems. A combination of the reference text and any one of the interface texts, as listed, is a perfectly good implementation of the virtual machine, and has been used as such. Equally, it is the blueprint for the various machine coded and microcoded implementations of the virtual machine which are detailed in the final sections of this manual.

Consult the section at the end of this manual which relates to your particular machine to discover which sources are provided in the distribution package for your machine.

Should you wish to implement the LispKit system afresh on a new machine on which there is available an implementation of Pascal, then the only change which you need make to the text of the reference machine, as listed in the companion volume to this manual, is to provide the definition parts of the following declarations.

```
const TopCell
procedure GetChar(var ch : char)
procedure PutChar(ch : char)
procedure Initialise(Version, SubVersion : char)
procedure Terminate
```

Consult the implementor's guide for a description of the required behaviour of each of these routines; the example codings provided in the listings volume should also be of assistance.

All of the LispKit Lisp sources in the companion volume are entirely machine independent, as are the object codes, so no changes would be necessary to these.

The implementor's guide

The later chapters of the book should be read by anyone who wants to understand or change the architecture of the LispKit implementation. This section of the manual deals only with the differences between this implementation and that described in the book.

Lazy evaluation: LDE, UPD and AP0

In the book, the lazy evaluation strategy is discussed in terms of the addition of explicit delaying and forcing operations to the Lisp program. The implementation of the language described in this manual automatically delays and forces any expressions which need to be so treated.

The lazy evaluation strategy requires that an expression be evaluated only when its value is needed in the evaluation of the whole program, and that once an expression has been evaluated, its value is noted so that it need not be recalculated. For this purpose, a new form of cell is added to the store of the machine — a recipe — which is represented as a pair of a type distinct from that of the cons cell. The head of a recipe is the code to be executed in order to evaluate the expression represented by the recipe, and the tail of the recipe is the environment to be used during the execution of that code. Three new intructions are added to the machine: LDE — load expression — which constructs a recipe; AP0 — apply to no arguments — which inspects the value at the top of the stack, and causes it to be executed if it is a recipe; and UPD — update recipe — which is used to return from the execution of a recipe.

The transition for the LDE instruction, opcode 22, using the notation of the book, is

    s e (LDE c'.c) d   →   ((c':e).s) e c d

where the pair (c':e) that is left on the top of the stack is marked as being a recipe. The effect of an LDE instruction is similar to that of an LDF instruction, and a recipe is treated very much like the closure for a function value. The essential difference is that a recipe contains the code and the whole of the environment in which that code is to be executed, whereas a closure still has one component of the environment missing — the argument list which is supplied by an AP instruction.

As the AP instruction causes the execution of the code in a closure, so there is an instruction, AP0 with opcode 24, which causes the execution of the code of a recipe:

    ((c':e').s) e (AP0.c) d   →   NIL e' c' (((c':e').s) e c.d)
    (a.s)   e (AP0.c) d   →   (a.s) e c d   if a is not a recipe

It differs from the AP instruction in two respects. Firstly, it tests whether the value at the top of the stack is in fact a recipe; if not, then the AP0 instruction has no effect. Secondly, when a closure is applied, the second item on the stack is an argument list which is added to the environment of the closure; when a recipe is forced, there is no argument list, so the environment for the execution of the code of the recipe is just that contained in the recipe itself. Notice also that the recipe is not removed from the top of the dumped stack, since it will be needed again, by the UPD instruction at the end of the recipe code.

In order to prevent repeated evaluation of recipes, an instruction is required which overwrites a recipe with its value, once this value is calculated. Since this overwriting must happen at the end of the execution of the code of a recipe, the operation is combined with the RTN operation, and is implemented by the UPD instruction, with opcode 23,

```
(v) e' (UPD) (s e c.d)  →  rplaca(s,v) e c d
```

This restores, from the dump register, the values of the registers which were current when the recipe was forced by an AP0 instruction, and overwrites the recipe, which is now at the top of the restored stack, with the value calculated by the code of the recipe.

The compiler has been modified from that described in the book to generate these instructions when it is necessary to delay the evaluation of an expression. The strategy adopted is to delay the evaluation of the arguments to each cons operation, the arguments to each lambda defined function, and correspondingly, the right hand sides of all definitions in let and letrec expressions. Again in the notation of the book, the changes to the compiler are described by:

```
(cons e1 e2)*n  =  (LDE e2*n|(UPD) LDE e1*n|(UPD) CONS)

(e e1 ··· ek)*n  =  (LDC NIL LDE ek*n|(UPD) CONS ···
                     LDE e1*n|(UPD) CONS e*n AP)

(let e (x1.e1) ··· (xk.ek))*n =
                     (LDC NIL LDE ek*n|(UPD) CONS ···
                      LDE e1*n|(UPD) CONS
                      LDF e*m|(RTN) AP)
     where m = ((x1 ··· xk).n)

(letrec e (x1.e1) ··· (xk.ek))*n =
                     (DUM LDC NIL LDE ek*m|(UPD) CONS ···
                      LDE e1*m (UPD) CONS
                      LDF e*m|(RTN) RAP)
     where m = ((x1 ··· xk).n)
```

Further, since an object loaded from the environment may turn out to be a recipe for the value which was expected, some AP0 instructions must be inserted into the code in those places where it may be necessary to force recipes, so for names of variables:

```
x*n = (LD location(x,n) AP0)
```

and since data structures may have delayed components:

```
(head e)*n  =  e*n | (CAR APO)
(tail e)*n  =  e*n | (CDR APO)
```

Interactive execution: STOP, READ and PRINT

The SECD machine described in the book first reads a program and its arguments, then executes that program to completion, and finally prints the result. The implementation of the language described in this implementation makes use of lazy evaluation, so can handle infinite data objects. This property is used to construct a notionally infinite stream of s-expressions which are read from the keyboard, and a notionally infinite stream of s-expressions which are output to the screen. Since a prompt requesting input from the user does not depend on the value that is input, the mechanism of lazy evaluation allows prompts to be issued to the screen, and for input from the keyboard to be taken, in their proper order.

The mechanism which constructs the recipe for the input stream is written in LispKit Lisp, and appears in the text of the bootstrap loader — LOADER.LSO — but requires the addition of an instruction to the machine. The transition for the READ instruction, opcode 25, is

```
s e (READ.c) d   →   (x.s) e c d
```

where x is the data structure represented by an s-expression newly read from the current input file. The compiler never generates this instruction, which is found in the function read_item in the library SECD_CODE. This function is used in the bootstrap loader — LOADER — which constructs a stream using the function

```
input = (lambda NIL (sequence item (cons item (input))))
              where   item = (read_item)
```

When this function is applied to an empty argument list, it first inspects the value of item, which causes a call of read_item to read an s-expression from the input. This having been done, it returns a pair consisting of that value which was read, and a recipe for another application of input. Thus, no matter in which order the components of the input list are inspected by the user, the nested calls of the function input inspect them in sequence, and they are read from the input in their proper sequence.

Similarly, there is an extra instruction, PRINT with opcode 26, which causes a representation the value at the top of the stack to sent to the output, and then discards that value. Its transition is

```
(v.s) e (PRINT.c) d   →   s e c d
```

and, like READ, is never generated by the compiler. It appears only in the function print_item in the library SECD_CODE. This function should be applied to one argument, and always returns that argument as its result, having first printed a representation of the argument. In the bootstrap loader there is defined a function

```
consume = (lambda (s) (step s))
where step = (lambda (s)
                (sequence (print_item (head s)) (consume (tail s))))
```

which causes calls of print_item to occur in the proper sequence to output
the components of the stream given as an argument to consume.

Finally, there is a change to the transition for the STOP instruction. If
the consume function in the loader were written exactly as above, then since
the tail call of consume would happen before the execution of the RTN
instruction in the outer call, more and more space would be consumed by
the unnecessary retention of old register values in the dump. Accordingly,
the tail call of step in the definition of consume is not compiled. Instead, a
data structure is returned from each call of consume representing the
application of step. This structure is simply a pair of the closure for step,
and the argument list. The STOP instruction, opcode 21, is modified so that
if it is executed with such a pair at the top of the stack, instead of
terminating it causes the application to happen. The transitions for STOP
are accordingly

```
(((c'.e').a).s) e (STOP) d   →   NIL (a.e') c' (s e (STOP).d)
(NIL.s)         e (STOP) d   →   (NIL.s) e (STOP) d
```

the former case representing an application, and the latter termination. The
STOP instruction is never generated by the compiler and appears only in
the function run_and_halt in the library SECD_CODE, and in the
program HALT.

Atom construction and decompostion:
CHR, IMPLODE and EXPLODE

At the time of writing this manual, there are two schemes in use for
construction and decomposition of atoms. Earlier SECD machines — with
version number up to 3a — have a single instruction, CHR, with opcode 27,
which constructs single character atoms. The transition for this instruction
is

```
(n.s) e (CHR.c) d   →   (an.s) e c d
```

where n is a number and an is an atom consisting of a single character with
ASCII character code n. The instruction is never generated by the compiler,
but is found in the function chr defined in the library SECD_CODE, by
which definition

```
(chr n)
```

evaluates to give the atom that prints as character n. Two of these atoms are
treated specially by 3a machines: chr of thirteen is printed as a line break;
the space character in chr of thirty two is not printed, so that printing this
atom causes only the inter-atomic space to be output.

In machines with version number 3b and later, the treatment of atoms has been much simplified. Instead of the separate storage spaces for atoms and cells which were described in the book, and used in earlier machines, the characters of atoms are now stored in the cell store of the SECD machine, and are accessible through a new machine register, the SymbolTable. There are also two instructions for the synthesis and analysis of atoms, namely IMPLODE with opcode 27, and EXPLODE with opcode 28. Their transitions are as follows:

```
(32.s) e (IMPLODE.c) d   →   (anull.s) e c d
```

where anull is the symbol containing no characters;

```
(n.s) e (IMPLODE.c) d   →   (an.s) e c d
```

where n is a number different from thirty two, and an is the symbol consisting of a single character whose ASCII code is n;

```
(ln.s) e (IMPLODE.c) d   →   (al.s) e c d
```

where ln is a list of numbers, and al is the symbol consisting of the sequence of characters with those ASCII codes;

```
(al.s) e (EXPLODE.c) d   →   (ln.s) e c d
```

where al is a symbol, and ln is the list of ASCII codes of the characters of that symbol. The two special cases in the transitions for IMPLODE should be considered as purely temporary measures to allow version 3a code to run on 3b machines. It is intended to define an additional two functions implode and explode to use these instructions, and replace the definition of chr so that

```
(chr (quote 32))  =  (implode (quote NIL))
(chr n)           =  (implode (list n)) otherwise
```

Again, at the time of writing, none of the LispKit Lisp code supplied in the distribution kits makes use of the IMPLODE and EXPLODE instructions, and the only source of CHR instructions is the use of the function chr.

Closures and the loading of code

In the book, the compiler is described by defining an operation * which takes a LispKit Lisp source and a syntactic environment and returns the code which must be executed to evauate that expression in that environment. This operation is implemented in the compiler itself by a function comp for which

```
e*n = (comp e n (quote NIL))
```

From the text of the compiler, it will be seen that the result returned by the compiler when applied to the source text ⟨exp⟩ is in fact

```
(⟨exp⟩*NIL . NIL)
```

This pair is intended to represent a closure, the closure for the function

```
f = (lambda NIL ⟨exp⟩)
```

This form is chosen because a finite, loop-free such closure may be found for any expression ⟨exp⟩ even if it has an infinite value. The closure, being finite, may be input and output at will, and is readily executed to yield the value of the expression ⟨exp⟩. This mechanism is also used in the hand compiled code in the library SECD_CODE, some functions of which are represented by quoted closures.

The bootstrapping mechanism in the LispKit machine expects the first item read from the input to be a closure, and the initial values of the registers are determined by that closure as follows:

```
let x = get_exp in
        begin S := NIL; E := tail(x); C := head(x); D := NIL end
```

For convenience of use, the initial input is taken from a particular named file in the filing system (dependent on the particular machine for which the implementation is intended) and this file normally contains the closure for the LOADER.

Subsequently, if a closure is read from the input stream, as for example by the loader, it may be treated as if it were the code for f above. For example, in the text of the bootstrap loader, the function execute is defined as

```
execute = (lambda (s) ((load_code (head s)) (tail s)))
        where    load_code = (lambda (s) (s))
```

Now, if (head s) is the closure emited by the compiler for the text ⟨exp⟩, then it is the closure

```
(⟨exp⟩*NIL . NIL)
```

so evaluating

```
(load_code (head s))
```

has the effect of leaving the value of ⟨exp⟩ at the top of the stack, and provided that ⟨exp⟩ is a function the application in the body of execute is an application of the function which is the value of ⟨exp⟩ to the remainder of the stream s.

The standard Pascal reference SECD machine

The portability of the LispKit system derives from its use of a small virtual machine which is easily implemented in any host environment. With the exception of this virtual machine, the whole of the system, and all user programs, can be carried from implementation to implementation with no changes being necessary.

In order to achieve the best performance possible, the virtual machine is usually tailored to each particular installation, for example, by microcoding the instruction execution. For this purpose, the Pascal text of

the reference machine should be taken as a guide to the required behaviour of any new implementations.

It is also possible to produce a perfectly functional implementation with minimal effort, by using the reference text directly, providing that an implementation of Pascal is available on the host machine. The reference text has the form of a Pascal program from which a number of definitions have been omitted. An implementor need only supply forms of these definitions which are appropriate to his particular host machine and operating system.

Potential implementors should consult the examples of machine specific codings in the companion volume to this manual, and read the section at the back of the manual on using these implementations. Within the limitations imposed by each host environment, the implementor's aim should be to produce a user interface as like those of existing implementations as is possible. The definitions which must be completed are described below.

const *TopCell*
TopCell is the number of storage cells which are available on the LispKit heap. The machine declares two arrays of integers, and three packed arrays of bits, each indexed from one to TopCell, so the value of TopCell will probably be limited by the amount of store available to the Pascal programmer.

procedure *GetChar(var ch : char)*
Successive calls to GetChar should read successive characters from the input stream, returning the character read in the argument variable. If the character code of the host machine is not ASC11, or some other ISO-7 code, then this routine should also translate the input characters into ISO-7. Calls of GetChar should be capable of returning at least the range of printable ISO-7 codes, and the code for a space.

Although calls of GetChar demand characters one at a time, keyboard input should be buffered a line at a time, giving the typist the opportunity to correct mistakes within the line. If this is not done by the operating system, then it should be done by the implementor of GetChar. A single space character should be returned by the call of GetChar at the end of each line, but no line terminating character should be returned.

There should be some mechanism for indicating that input from the keyboard has ended. This is often provided by the host operating system, but if not, should be implemented in GetChar. Conventionally, the typing of 'control Z' marks the end of keyboard input, and with it, the end of a line.

Whenever the file from which the current input is drawn becomes exhausted, a call of GetChar should prompt at the console for a new file, and attach the input stream to the new file. That particular call of GetChar should, however, return a space character, leaving the first character of the file to be returned by the next call.

Similarly, there should be a character, conventionally 'control Y', which, if it appears on a line of typed input, causes the output stream to be redirected. The call of GetChar which would have returned this character should instead prompt at the console for the name of a new output file, and attach the output stream to that file. The special character, and any subsequent characters on the line, should be discarded, and the call of GetChar should return a space character in its argument variable, denoting the end of the line.

For example, if the user types

> *a b ⟨ctrl Y⟩ c d ⟨return⟩ e f ⟨return⟩ g h ⟨ctrl Z⟩*

then successive calls of GetChar should return the codes for

> a b ⟨space⟩ e f ⟨space⟩ g h ⟨space⟩

and the third call should prompt

> Send output to where?

and the ninth,

> Take input from where?

The next call should read the first character from the file specified in response to this prompt.

procedure *PutChar(ch : char)*
Successive calls to PutChar should write their arguments as successive characters of the output stream. As with GetChar, if the host machine does not use an ISO-7 character code, then PutChar should take care of the translation. PutChar must be capable of outputting any printable ISO-7 character, and space. In addition, character code thirteen, ISO-7 carriage return, should cause a new line to be taken in the current output file.

procedure *Initialise(Version, SubVersion : char)*
This routine is called once, before any other user provided code is executed. It is responsible for the initialisation of the input and output streams, and for any initialisation which the user provided routines may require in a particular implementation. Specifically, every implementation of Initialise must do the following.

A message should be sent to the console, including the name of the implemenation, and the two characters Version and SubVersion.

If a bootstrap file exists, then the input stream should be arranged to start with the bootstrap file; if the bootstrap file does not exist, or cannot be read for some other reason, then a message to that effect should be sent to the console, and the user should be prompted for an input file in the normal way.

The output stream should be directed to the screen of the user's console.

Notice that the files 'input', and 'output' are provided in the program heading to be used for conducting the interaction with the user; 'InFile' and

'OutFile' are provided for use in the input and output streams. Neither of this latter pair is used anywhere in the reference text, so are free to be used as required by GetChar and PutChar.

procedure *Terminate*

This procedure is called once to signal the end of the execution of the machine. It is responsible for ensuring that any output buffering is flushed, that the current output file is established, and that any required machine specific finalisation is done. The call of Terminate should not return to its caller; rather it should relinquish control to the host operating system. Notice the provision of label 99 at the end of the reference text, should this be required.

Variations between machines

This section describes the method of installing the LispKit system on each of the machines for which support is provided by the authors of this manual. Here also are the details of the interface to the host operating system, and the machine specific performance parameters of each implementation of the LispKit virtual machine.

.

ICL Perq machines

The distribution floppy disk is written using the FLOPPY command of POS, and contains a command file for use with FLOPPY to retrieve distribution software. Unless the label on the floppy disk specifies otherwise, the disk is written single-sided and double-density.

To retrieve the distribution software, place the floppy disk in the Perq's drive, set the path-name of the directory to which the distribution is to be copied, and type the following sequence of commands

```
FLOPPY⟨return⟩
get get.cmd⟨return⟩
@get⟨return⟩
q⟨return⟩
```

The entire contents of the floppy disk have now been copied into the current directory.

The POS Perq distribution contains two Pascal programs which implement the virtual machine, one called SECD which is written entirely in Pascal, and one called fSECD (f for fast) which is partly coded in Perq microcode. The microcode appears in the files SEGMP.MICRO (source) and SEGMP.BIN (object). Each of these Pascal programs must be compiled and linked, so type the following commands

```
Compile SECD⟨return⟩
Link SECD⟨return⟩
Compile fSECD⟨return⟩
Link fSECD⟨return⟩
```

It is now possible to execute either machine by typing its name as a command, so a short interaction might appear as

```
⟩ fSECD⟨return⟩
Perq microcoded SECD machine 3a

Take input from where? NFIB.LOB⟨return⟩

Take input from where? CONSOLE:⟨return⟩
15⟨return⟩
1973 ⟨ctrl Z⟩

Take input from where? HALT.LOB

⟩
```

The machine which is normally used is fSECD, which is some twenty times faster than SECD. The latter machine is included because its text gives a complete description of the virtual machine, and because it is simpler to

modify should you wish to experiment with different designs for the virtual machine.

Normally, when you execute SECD or fSECD, the file SECD.BOOT should appear in the current directory, or in some other directory in your search list, since it contains the default system LOADER which is used to bootstrap the virtual machine. Similarly, SEGMP.BIN should also be available during the execution of fSECD.

The convention for file names is as used in the body of the manual, that is a program such as NFIB is described by source text in a file

NFIB.LSO

which is compiled to produce the code (closure) contained in

NFIB.CLS

Code files with extension .LOB, such as

NFIB.LOB

contain code or sequences of codes which may be executed by the default system LOADER. Finally, files with extension .LIB, such as

STANDARD.LIB

contain association lists (libraries). Any valid form of POS file name may be used in response to prompts from the virtual machine, including full path names, and all abbreviations supported by POS, for example

part:user⟩dir1⟩dir2⟩f
..⟩f⟩g
:user⟩dir⟩..⟩a⟩b

The various performance parameters for the two machines are:

	SECD	fSECD
User function calls per second	75	2k
Number of cells available	10k	32k
Precision of arithmetic	16 bits	16 bits
Cell store garbage collector	recursive	recursive
Boot file name	SECD.BOOT	SECD.BOOT
Key to end file	control-Z	control-Z
Key to redirect output	control-Y	control-Y
Key to interrupt machine	control-shift-C	control-shift-C
Key to suspend output	control-S	control-S
Key to resume output	control-Q	control-Q

68000 machines

Unless the label specifies differently, the distribution floppy disk is a UCSD p-system floppy disk written in the (default) Sage 80-track format and contains a 1280 block UCSD volume called DISTRIB:. This volume contains a number of subsidiary volumes which, in turn, contain the files of the distribution. In order to use the system, you will need a copy of Version IV of the UCSD p-system. The distribution software does not make use of any Version IV specific features of the Pascal system, but the code files were all created with Version IV software. It should be possible to recompile and reassemble the texts, all of which are provided, should you be running any other Version of the p-system. The distribution disk was created with Version IV.12.

Two versions of the LispKit virtual machine are provided in the subsidiary volume VM:. One is a Pascal source, called the reference machine,

VM:SAGE.SECD.TEXT

which runs very slowly, and has a small cell space. You will need to compile this source should you want to run the reference machine. It is included in the distribution since it describes in detail the behaviour of the virtual machine, and since it is easily modified should you wish to experiment with the architecture of the virtual machine.

The other version is coded in 68000 assembly code, and it comprises all of the other text files in the VM: volume.

VM:SECD.HOST.TEXT

is a Pascal host. and

VM:SECD.GUEST.TEXT

a 68000 code external procedure, which have been compiled and assembled, respectively, and linked to form the code file

VM:SECD.CODE

which is the code to be executed to run the virtual machine.

To run the system, you will require to make a copy of the system bootstrap loader, which is contained in the file

VM:SECD.BOOT

on the distribution disk. and to place it on your system disk, under the name

*SECD.BOOT

Note that since SECD.CODE uses all of the store of the Sage for its cell store, the system disk should not be a RAM disk, since this may be

overwritten before the bootstrap file is read. Similarly, if you are using a RAM disk, then any files on it should be backed up to a non-volatile medium before executing SECD.CODE.

The suggested procedure for getting the sytsem up is as follows. First, configure a p-system system disk which does not boot from the RAM disk; then, using the p-system filer, mount the subsidiary volume VM: from the distribution disk, and copy the files SECD.CODE and SECD.BOOT to the new system disk, by typing

> *fomDISTRIB:VM.SVOL⟨return⟩*
> *tVM:SECD.CODE,*$⟨return⟩*
> *tVM:SECD.BOOT,*$⟨return⟩*

Now, copy the subsidiary volumes LSO:, LIB:, CLS:, and LOB: from the distribution disk to a working disk, place this on-line, and mount each of these four subsidiary volumes.

It is not necessary to copy the LispKit files, since the distribution disk will make a perfectly adequate working disk, but you will have more disk space available if you do not have the texts of the virtual machines on-line.

The machine is now executed by typing

> *x*SECD⟨return⟩*

so a short interaction might appear as

> Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? *p*
> Prefix file names by what volume? *LISPKIT:*
>
> Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? *q*
>
> Command: E(dit, R(un, F(ile, C(omp, L(ink, X(ecute, A(ssem, D(ebug,? *x*
> Execute what file? **SECD⟨return⟩*
> SECD machine version 3b for the Sage ij/iv
>
> Take input from where? *:LOB:NFIB⟨return⟩*
>
> Take input from where? *CONSOLE:⟨return⟩*
> *15⟨return⟩*
> 1973 *⟨ctrl Z⟩*
>
> Take input from where? *:LOB:HALT*
>
> SECD machine terminated normally
>
> UCSD p-System IV.1 BootStrap

The convention for file names is different from that used in the body of the manual, since there is a small limit (seventy-seven) on the number of files that may appear in any on p-system directory. The file system is still the p-system file system, but rather than use the form

> ⟨filename⟩.⟨extension⟩

each type of LispKit file is kept in a separate volume, and the file would appear (in p-system terms) as a file called ⟨filename⟩ in a susidiary volume called ⟨extension⟩, that is

⟨extension⟩:⟨filename⟩

so, for example, NFIB is described by source text in a file on the LSO: subsidiary volume on the distribution disk, that is

LSO:NFIB

in p-system terms. Files must be referred to in this way when using the Pascal reference machine.

It is not possible to 'mount' subsidiary volumes in the LispKit machine, so a slightly different file name convention is adopted. Simple volume names

CONSOLE:
PRINTER:
REMOTE:
REMIN:
REMOUT:

are used to refer to serial devices. Names of the form

⟨volume⟩:⟨file⟩

refer to a file called ⟨file⟩ on a volume called ⟨volume⟩ which must be one of unit 4, or unit 5 (normally the left or right hand floppy disk drives on the Sage ij, or the sole floppy disk drive on a Sage iv), or units 9, 10 or 12 (normally the three accessible partitiions of the Winchester disk on a Sage iv). Names of the form

⟨volume⟩:⟨subsidiary⟩:⟨file⟩

refer to a file called ⟨file⟩ on a subsidiary volume called ⟨subsidiary⟩. The subsidiary volume must be in a file called

⟨subsidiary⟩.SVOL

on the volume ⟨volume⟩ which must, again, be in unit 4, 5, 9, 10 or 12. The following abbreviations are allowed, following the p-system style: an empty file name (just ⟨*return*⟩) stands for CONSOLE:, a file name with no ':' is prefixed with the prefix volume name, a file name beginning ':' is prefixed with the prefix volume name, a file name beginning '*' is prefixed with the system volume name. There is no provision for setting of the prefix or system names, each of which is inherited from the p-system. Similarly, the date used to create new files is that set in the p-system before executing the machine. The BIOS configuration — remote line speed, type of printer, etc. — however, is that in the BIOS file when the system was last booted.

For examples of these file name conventions, if the working disk is called LISPKIT:, and is the prefix disk, then the program NFIB may be found in a file which is referred to as either one of

LISPKIT:LSO:NFIB   or   :LSO:NFIB

which is compiled to produce the code (closure) contained in the file called either

LISPKIT:CLS:NFIB   or   :CLS:NFIB

Code files on the subsidiary volume LOB:, such as

LISPKIT:LOB:NFIB   alias   :LOB:NFIB

contain code or sequences of codes which may be executed by the default system LOADER. Finally, files in the volume LIB:, such as the file called by either of

LISPKIT:LIB:STANDARD   or   :LIB:STANDARD

contain association lists (libraries). In common with the p-system, each volume name component of a file name must be at least one character long, and no more than seven; similarly, the final component must be at least one character long, and no more than fifteen. Case of characters is not significant.

Note also that all files written with this machine are p-system Datafiles, irrespective of any extension on the file name. If you want to process LispKit output with a program that expects Textfile input, the LispKit output must be reformatted by the program

VM:DATA — >TEXT

which copies its input to its output, making the necessary changes of format. A typical run would be

```
xVM:data − >TEXT pi = LSO:NFIB po = *NFIB.TEXT
<ctrl Z>
```

which produces a text copy of the LispKit source of NFIB on the system disk.

The various performance parameters for the two machines are:

	Pascal	Assembler coded
User function calls per second	7	1k
Number of cells available	4k	*
Number of symbols available	200	500
Precision of arithmetic	16 bits	32 bits
Cell store garbage collector	recursive	non-recursive
Boot file name	*SECD.BOOT	*SECD.BOOT
Key to end file	control-Z	control-Z
Key to redirect output	control-Y	control-Y
Key to interrupt machine	set by p-system	control-B
Key to suspend output	set by p-system	control-S
Key to resume output	set by p-system	control-S

* The number of cells available in the assembler coded machine depends on the size of the installed RAM. In a half megabyte Sage ij, there are some sixty thousand cells available, the exact number depending on the size of the BIOS which is used.

Digital Equipment VAX machines

The LispKit system software is supplied on magnetic tape, and unless the label specifies otherwise this is a nine track, sixteen hundred bits per inch, phase encoded tape, primarily intended for use on Digital Equipment VAX machines running VMS. The tape is written using the COPY operation of VMS, and conforms to the ANSI standard. We trust that this will make it possible for users with other host machines to import the LispKit system using this VAX tape.

The recommended procedure for recovering the distribution files on a VMS VAX (subject to local custom and practice at your site) is to mount the tape on tape drive ⟨*xn*⟩, select a directory ⟨*directory*⟩ to receive the files, and to execute the following DCL commands:

```
$ Define LKTape MT⟨xn⟩:
$ Define LKDisk ⟨directory⟩
$ Allocate LKTape
$ Mount LKTape
$ Copy LKTape:*.*; LKDisk:*.*
```

The entire contents of the tape have now been copied into the selected directory.

The VMS VAX distribution contains two programs which implement the virtual machine: one is called SECD and is written entirely in Pascal; the other is called fSECD (f for fast) and is largely coded in VAX machine code. Since only texts are supplied on the magnetic tape, you will need to compile, assemble, and link these programs, as follows:

```
$ Pascal LKDisk:SECD
$ Link LKDisk:SECD
$ Pascal LKDisk:fSECD
$ Macro LKDisk:fSECDasm
$ Link LKDisk:fSECD,fSECDasm
```

See the file MAKESECD.COM on the tape for the commands used to construct the original system. It is now possible to run either machine by typing either

```
$ Run SECD
```

or

```
$ Run fSECD
```

as appropriate. It is suggested, however, that you construct command files

of the following form, which establish a number of useful logical names:

```
$ Create 〈command file name〉
$ Define LK 〈directory〉
$ Define LispKit$SECDboot LK:LOADER.CLS
$ Run LK:fSECD
$ DeAssign LispKit$SECDboot
$ DeAssign LK
〈ctrl Z〉
```

which can either be executed by typing

```
$ @〈command file name〉
```

or by defining a symbol, say

```
$ SECD := = @〈command file name〉
```

so that you can invoke either machine by typing its name as a command. See the file SECDPREP.COM on the tape for the commands used to run the original system. A short interaction might appear as

```
$ fSECD
VAX Pascal SECD machine 3a
SECD machine 3a implementation for the Vax (test)


Take input from where? LK:NFIB.LOB〈return〉

Take input from where? CONSOLE:〈return〉

15〈return〉
1973 〈ctrl Z〉

Take input from where? LK:HALT.LOB〈return〉


SECD machine terminated normally


$
```

The machine which is normally used is fSECD, which is faster than SECD. The latter machine is included because its text gives a complete description of the virtual machine, and because it is simpler to modify should you wish to exeriment with different designs for the virtual machine. Note that at the time of writing this manual, the tape being distributed includes a version 3b SECD machine, but a version 3a fSECD machine.

Normally, when you execute SECD or fSECD, the logical name

```
LispKit$SECDboot
```

should be associated with the file

```
LOADER.CLS
```

copied from the distribution tape. This file contains the default system LOADER which is used to bootstrap the virtual machine.

Because not all of the file names used in the body of the manual are permissible VMS file names, a systematic name translation has been carried out in producing the magnetic tape: all underscore characters are omitted; the unextended name is then truncated after the ninth character. Thus the file referred to in the manual as

MAP_UNTIL_END.LSO

has been copied from the tape as

MAPUNTILE.LSO

and so on. Again, since several users may be sharing the distributed copy of the LispKit system at your site, we suggest that a user normally writes his own files in a directory which is selected as his default directory, and that all users share access to the copy of the tape. If you use the command files suggested above, then the logical name LK: can be used to refer to the distribution software, thus

LK:MAPUNTILE.LSO

With these provisions, the convention for file names is as used in the body of the manual, that is a program such as NFIB is described by source text in a file

LK:NFIB.LSO

which is compiled to produce the code (closure) contained in

LK:NFIB.CLS

Code files with extension .LOB, such as

LK:NFIB.LOB

contain code or sequences of codes which may be executed by the default system LOADER. Finally, files with extension .LIB, such as

LK:STANDARD.LIB

contain association lists (libraries). Any valid form of VMS file specification may be used in response to prompts from the virtual machine, including full file names, and all abbreviations supported by VMS, including logical names.

The various performance parameters for the two machines are:

	SECD	fSECD
User function calls per second	200	900
Number of cells available	40k	40k
Number of symbols available	500	ⁱ
Length of symbolic atoms	12 characters	ⁱ
Precision of arithmetic	32 bits	32 bits
Cell store garbage collector	recursive	non-recursive
Symbol store garbage collector	none	ⁱ
Boot file logical name	LispKit$SECDboot	LispKit$SECDboot
Key to end file	control-Z	control-Z
Key to redirect output	control-H	control-H
Key to interrupt machine	control-Y	control-Y
Key to suspend output	control-S	control-S
Key to resume output	control-Q	control-Q

ⁱ Since fSECD is a 3b machine, the atoms are stored in the cell store, and are of unlimited length and number, cell store permitting, and symbols are subject to the normal cell store garbage collection regime.

Oxford University Computing Laboratory
Programming Research Group Technical Monographs
Autumn 1983

This manual is one of a series of technical monographs on topics in the field of computation. Copies may be obtained from:

Programming Research Group, *(Technical Monographs)*
8-11, Keble Road, Oxford.   OX1 3QD.   England.