

Contents

1	Introduction	1
I	User Manual	2
2	Getting Started	2
3	The Language	3
3.1	Basic Elements	3
3.2	Function Applications	5
3.3	Function Definitions	5
3.3.1	Lambda Expressions	6
3.3.2	Expressions	7
3.3.3	Constant Expressions	7
3.3.4	Conditional Expressions	8
3.3.5	Auxiliary Expressions	8
3.4	Recursive Definitions	9
3.5	Higher Order Functions	10
3.6	Delayed Evaluation	11
4	System Directives and Primitives	12
4.1	Directives	12
4.2	Primitive Constants and Functions	13
4.2.1	Constants	13
4.2.2	Arithmetical Functions	14
4.2.3	Trigonometrical Functions	16
4.2.4	Predicates	18
4.2.5	Relational and Logical Functions	19
4.2.6	S-expression Structure Function	20
4.2.7	Environment Functions	20
4.2.8	Delayed Evaluation Functions	21
4.2.9	Stream Processing Functions	21
4.2.10	Miscellaneous	22
II	Implementer Manual	23
5	System Structure	23
5.1	User Interface	23
5.2	Compiler	24
5.3	Emulator	28

<i>CONTENTS</i>	2
6 Conclusion	30
7 Acknowledgements	31
8 References	32
A Source Listing	33
B Sample Programs	68
Index	70

Waterloo Unix Prolog and Lisp Environment

Mantis H.M. Cheng

Logic Programming and Artificial Intelligence Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada. N2L 3G1

May 27, 2001

Abstract

This is a report on WUPLE a functional programming language written in Waterloo Unix Prolog (WUP). The implementation is based on the work of Lispkit Lisp by Peter Henderson [4]. WUPLE extends Lispkit Lisp in several ways: 1) it is an interactive programming environment integrated with WUP; 2) it has a compile-time macro expansion facility; 3) it supports incremental programming much like many other LISP programming systems; 4) it has an improved and uniform treatment of system-defined and user-defined functions. WUPLE is a full implementation of Lispkit Lisp which supports static-binding, closures and delayed evaluation. Functions are regarded as first-class objects and can be used like any simple values. This report is divided into two parts: part I, the User Manual, describes the language and the environment; part II, the Implementer Manual, discusses the extensions and modifications to the original Lispkit Lisp implementation. The conclusion summarises the various advantages of using Prolog as the implementation language and indicates our future research direction.

1 Introduction

This document describes a LISP-like programming system [6, 7] built on top of the Waterloo Unix Prolog environment [3] (version 2.0). Waterloo Unix Prolog and Lisp Environment, hereafter called WUPLE, is based on the language Lispkit Lisp described in the book *Functional Programming: Application and Implementation* by Henderson [4]. WUPLE is a full implementation of Lispkit Lisp with some extensions to include some features to support Scheme programming style. Scheme, a lexically scoped dialect of LISP which supports closures and higher order functions, is designed and implemented at MIT [1]. Lispkit Lisp supports programming in a purely functional style, and stream processing by explicit delayed evaluation. WUPLE is designed and implemented as a tool to study the style and techniques of functional programming. It is also used to demonstrate the possibility of implementing a usable functional programming system in Prolog and to investigate our ultimate goal of amalgamating functional and logic programming. The description given in this document assumes no knowledge of Waterloo Unix Prolog (WUP) environment. The reader is referred to the document *Waterloo Unix Prolog User's Manual* by Maarten van Emden and Randy Goebel for details about WUP [10]

This document is divided into six sections. Next section, Section 2, describes briefly some basic environment features of WUPLE. Section 3 gives the syntactic and semantic definitions of the input language. Section 4 provides a more detailed look at the command language and primitive functions of this system. Section 5 is written for those interested in the internal structure of WUPLE; it introduces different components of this system and highlights some of the implementation techniques. Section 6 concludes with a few remarks on the current status of WUPLE and its future development. The Prolog source listing and some sample WUPLE programs are added at the end as the appendices.

Part I

User Manual

2 Getting Started

WUPLE resides as a program module [cf. WUP] in the program library. It is used as a normal user module and has only one accessible predicate, `wuple`, which the user can invoke inside WUP. Once it is started, it prints the following messages:

```
WUPLE (version 1)
Type '(help)' for help.
[0]->
```

The last message “[0]->” is the system top level prompt. The number in square brackets is the number of closing parentheses needed to complete the current input expression, e.g.,

```
[0]->( + ( * 3
[2]->      2 ) 1 )
7                                     % result of the evaluation
[0]->                                     % return to the top level
```

If you type “(help)”, you get the following messages:

```
[0]->(help)
(def [rec] <named-exp> <expression>)
(dictionary)
(display '<function>')
(edit '<filement>')
(error <integer>)
(exit)
(load '<filename>')
(macro <named-exp> <expression>)
(undef <function>)
(wup)
```

Every input to the system must be an S-expression (symbolic expression), which is similar to many standard LISP programming systems; those already familiar with LISP should have no difficulty. We shall give a more detailed description of the input syntax in Section 3. To temporarily return to WUP, you can type “(wup)”. Within WUP, you can type the goal “?wuple” to get back in WUPLE. (Note: when you get an exception or when you hit the *break* key, you are automatically thrown back to WUP.) The system function `exit` allows one to leave the system.

If you have a file of programs, you can load it into the system by using the `load` function. While you are inside WUPLE, you can edit a file by calling the function `edit`. To view the definition of a certain function, say `F`, you can call the `display` function which returns the expression representing `F`. The function `dictionary` returns a list of names of all the functions that are currently stored in the system. You can define new functions using the system directive (i.e., they are not functions in the usual sense) `def` to add a new function definition into the program database (a place where all the currently accessible functions are stored). The keyword `rec` is used when you define a function which refers to itself in its body. A more detailed description of the system directives and functions is given in Section 4.

3 The Language

The syntax of the input language is an extension of the original concrete syntax of Lispkit Lisp. The extension introduces a slightly more readable form and is adopted from Scheme, so Scheme programs with minor modifications can be made to run on this system.

WUPLE is a purely functional programming system; there are no side-effects or destructive assignments. Programs are constructed solely by combining functions. Functions are treated as first-class objects, i.e., they can be manipulated as simple constant objects. (Note: conventional LISP implementations do not—or do with great difficulty—allow users to define higher order functions.) WUPLE promotes programming at the functional level (functions as values) rather than just the object level (constants as values).

In the following subsections, we describe what the basic data types are, how to define and construct new functions from the existing ones, and how to use delayed evaluation to build infinite sequences.

3.1 Basic Elements

This version of WUPLE is still experimental. It is not intended to be used for writing large application programs. Its primary goal is to teach the ideas about functional programming. The basic data types supported are: integer, float (floating point number) and symbol. Pair and list allow one to build compound data structures. We now give a more formal description of our input language in terms of BNF and regular expressions.

S-expression ::= atom | pair | list

An S-expression is either an atom, a pair or a list. These objects form the basic elements (or tokens) of our language. Any one of these basic elements must be delimited by one or more spaces or by parentheses. Precisely, they are defined as:

atom ::= number | symbol
number ::= integer | float
integer ::= [sign] digit {digit}
float ::= [sign] digit {digit} ‘.’ {digit} [exponent]

`exponent ::= ('E' | 'e') [sign] digit {digit}`
`sign ::= '+' | '-'`
`symbol ::= any sequence of printable characters which does not`
 begin with a digit and does not include parentheses.

`[]` means optional.

`{}` means zero or more occurrences.

Note: a comment begins with a `'%'` or `';` and continues to the end of line.

e.g., valid integers: 123 -123 +123
 valid floats: 1.23 -1.23E3 +123.0e-3
 valid symbols: abc ++ b6!d

Every letter is case sensitive (i.e., upper and lower case letters are distinct). A **symbol** is equivalent to a variable in most conventional programming languages; it is used to denote something which has a value. To make a symbol stand for itself, we have to “quote” (or just `'`) that symbol. e.g., `(quote abc)` or `'abc` stands for the symbol “abc”. `''abc` is the same as `(quote (quote abc))` which has as value the expression `(quote abc)`. (Note: every number is implicitly quoted; the expression `(quote 1)` has the same value as the integer 1.)

`pair ::= ('S-expression '.' S-expression')`
`list ::= ('S-expression '.' list') | '()' | 'nil'`

A **pair** consists of two **S-expression** enclosed by parentheses and separated by a `'.'`. Note that white spaces are needed to separate the two **S-expressions** and the `'.'`. The left part of a **pair** represents the “car” (or the head) field and right part represents the “cdr” (or the tail, pronounce as cudder) field. A **list** is a special nested type of **pairs** which always ends with the empty list constant `'()` (or `nil`).

e.g., valid pair: (a . b) ;; not a list
 valid list: (a b c)

The list `(a b c)` is the same as the pair `(a . (b . (c . nil)))`. The “cdr” field of a **list** is always a **list**, but it is not necessarily so for a **pair**. In defining an **S-expression**, the parentheses are **not** used for grouping; their existence is significant. The **S-expressions** `(a)` and `a` denote two different objects.

This completes our description of the syntax of our input language. For the system to understand its input, it must first define what constitutes a valid program or function. In the next two sections, we describe how to construct such a program from the above given basic elements.

3.2 Function Applications

A function is evaluated when it is applied to its arguments (or actual parameters). The result of applying a function can be either a constant or a function. In the next two sections, we are only concerned with functions computing simple values, e.g., numbers or lists. The discussion of functions taking functions as arguments and/or returning functions as values will be left until Section 3.5 on higher order functions. Let us first consider how to “execute” a function.

WUPLE is an interactive system. Everything you type is either a function definition, which begins with the keyword `def`, or a function application. For example, to apply the primitive function “+” to some arguments, you type:

```
[0]->( + 1 2)
3
[0]->( + (* 2 3) (* 4 5))    ;; 2*3 + 4*5
26
```

Hence, every input expression which does not begin with the keyword `def` is a function application (or an applicative expression). Every applicative expression has the form:

```
applicative-expression ::= combination | symbol | lambda-expression
combination             ::= ‘(operator operand)’
operator                ::= applicative-expression
operand                 ::= {applicative-expression}
```

The **operator** is either the name of a predefined function, a lambda expression (more on this later), or another function application which returns as result a function. A function is evaluated by first evaluating its **operands** (or arguments)—from *right to left*—and then applying the function **operator** to these arguments. The value of an **applicative-expression** depends only on the values of its operands and is independent of the order in which it is evaluated; this is a property of an applicative (or functional) language. WUPLE uses an *applicative* order evaluation strategy, i.e., it evaluates the **operands** first before applying the **operator**.

WUPLE supports static binding—a symbol must be defined before it is referenced. Since there are no type declarations in WUPLE, it is the programmer’s responsibility to ensure that the arguments are valid before applying a function. Misuse of system primitive functions causes the “illegal input?” message.

3.3 Function Definitions

A function can be defined in two ways: by a lambda expression, or by the `def` construct. A function defined by the `def` construct is automatically added into the program database and can be referred to later by its name. A lambda expression defines an unnamed expression which denotes a function. The following two statements define the same function:


```
(lambda (x) (* x x))      ;; (1)
(def (square x) (* x x))  ;; (2)
```

The statement (1) is a lambda expression; it defines a function which takes one argument (or formal parameter) and multiplies the argument by itself. The statement (2) defines a function `square` with similar operation. To compute the square of a number, we can use either one of the above constructs:

```
[0]->((lambda (x) (* x x)) 3)
9
[0]->(square 3)
9
```

In the first case, the operator is a lambda expression and the operand is 3. In the second case, the operator is an identifier `square`, which evaluates to a function defined in (2), and its operand is also 3. A definition (or `def` construct) is defined to be:

```
definition      ::= '(‘def’ named-expression body ‘)’
named-expression ::= symbol | ‘(‘symbol parameter‘)’
parameter       ::= {symbol}
body            ::= expression
```

```
e.g., (def x 1)                ;; (3)
      (def (succ n) (+ n 1))    ;; (4)
```

The definition (3) defines `x` to be the constant 1. The definition (4) defines the successor function `succ`. These have the side-effect of adding two new definitions, `x` and `succ`, into the program database for later use. The name of a definition is given by the first symbol appearing in the `named-expression`. The term `parameter` stands for zero or more symbols following the definition name. In fact, every function defined using the `def` construct is translated into a binding tuple, i.e., a name and its value which is an expression. For example, the body of definition (4) is translated to a lambda expression:

```
(def succ (lambda (n) (+ n 1)))
```

and so the value of `succ` is a lambda expression— which can be read as $succ = \lambda n.(n + 1)$.

3.3.1 Lambda Expressions

Formally, a lambda expression is defined as:

```
lambda-expression ::= '(‘lambda’ ‘(‘parameter‘)’ body‘)‘
```

Every lambda-expression must begin with the keyword `lambda` and is enclosed by parentheses. The term **parameter** defines a list of formal parameters—which may be empty. The **body** is an expression. Every symbol in the **body** which is unquoted appearing in the **parameter** is a bound variable, otherwise it is a free variable and must get its value from its external environments, e.g., the program database. (Note: an environment is a list of associations between names and their values.) An error occurs if a free variable is undefined in its external environments.

Every lambda expression stands for an unnamed function. Two functions defined by two lambda expressions are two distinct functions even though they might compute the same function. The advantage of using the `def` construct to define a function over an unnamed lambda expression construct is that we do not have to re-type its definition every time we use it. Since a function is referred to by its name only (not by its arity—number of arguments it takes), it is removed from the program database whenever a new definition with the same name is added. Another major difference between these two styles of function definition is that an unnamed lambda expression is inaccessible outside the expression in which it is defined, while a function stored in the program database is global. The lifetime of the latter definition lasts until either a new definition is added, is removed explicitly by the `undef` [cf. Section 4] directive, or the user terminates his session. We can say that the program database provides the user an global environment which keeps all the initial bindings between names and values.

3.3.2 Expressions

An expression is defined as follows:

$$\text{expression} ::= \text{applicative-expression} \mid \text{constant-expression} \mid \\ \text{conditional-expression} \mid \text{auxiliary-expression}$$

The evaluation of an **expression** always returns a value which is either a constant or a function. The value of a lambda expression is an unnamed function; the value of an applicative expression is the value of the function application; and the value of a constant expression is always a constant.

3.3.3 Constant Expressions

A constant expression is defined as (note: see Section 4 on primitive constants for detail):

$$\text{constant-expression} ::= \text{number} \mid \text{truth-value} \mid \text{'('quote' S-expression)'} \\ \text{truth-value} ::= \text{'true'} \mid \text{'false'}$$

e.g., 123 true false '()' 'abc (quote abc)

3.3.4 Conditional Expressions

A conditional expression allows one to select a value from several expressions based on some given conditions. There are basically two ways of defining conditional expressions.

```
conditional-expression ::= cond-expression | if-expression
cond-expression       ::= '(cond clause {clause})'
clause                ::= '(predicate expression)'
```

```
e.g.,      (cond      ((> n 0)   E1)
                ((= n 0)   E2)
                ((< n 0)   E3))
```

The value of this `cond-expression` is the value of `E1` if `n` is greater than 0, `E2` if `n` is equal to 0, or `E3` if `n` is less than 0. The `predicate` in the last clause can be replaced by `else` or `true`, and the value of the last expression is always selected if none of the previous clauses succeeds. The `cond-expression` is the most general form of a conditional expression. The evaluation of the clauses in a `cond-expression` is done sequentially; the value of the expression of the first clause whose predicate evaluates to `true` is the value of the `cond-expression`. The other form of a conditional expression is defined as:

```
if-expression ::= '(if predicate then-part else-part)'
then-part    ::= expression
else-part    ::= expression
```

```
e.g.,      (if (> n 0) E1 E2)
```

The value of this `if-expression` is the value of `E1` if `n` is greater than 0, or `E2` otherwise. An `if-expression` is equivalent to a `cond-expression` with exactly two clauses. A predicate in a conditional expression can be any boolean function. The value of any boolean (or truth valued) function must be a constant which is either `true` or `false`. (Note: it is different from LISP which treats the empty list as the truth value `false` and treats everything else as the truth value `true`.)

3.3.5 Auxiliary Expressions

An auxiliary expression allows one to define an expression which has local bindings; it is similar to the block construct in block-structured programming languages. These local bindings are only accessible within the block and are removed at block exit. An auxiliary expression is defined to be:

```
auxiliary-expression ::= '(let binding {binding} body)'
binding              ::= '(named-expression expression)'
```

e.g.,

```
(let ((f y) (+ y 1)) ;; binding
      ((h x) (* x x)) ;; binding
      (* (f n) (h n))) ;; body expression
```

The value of an **auxiliary-expression** is the value of the **body** expression evaluated in the local binding environment plus the external environments. For the above example, suppose there is a function **f** defined in the external environments; the function **f** in the **body** is bound to the local function **f**, and not the one defined in the external environments. Both the functions **f** and **h** are local to this **let** block and are inaccessible from other functions.

3.4 Recursive Definitions

We mentioned earlier that a function must be defined before its use; how then do we define a recursive function? In many block-structured and static-binding programming language implementations, the compiler assume that a call to a procedure in which it is invoked is recursive and binds the call to the procedure itself somehow. This is against the rule of static-binding. The same case applies here.

To resolve this problem, we introduce the fixed-point operator **Y** [2, 5] which is defined as follows:

$$\mathbf{def} \mathbf{Y} = \lambda F.F(\mathbf{Y}(F)) \quad ;; (5)$$

The definition (5) says that the fixed-point operator **Y** applied to a function F is equal to the function F applied to the original application. Given a recursive definition:

$$\mathbf{def} f = \lambda x.E \quad \text{where } E \text{ contains } f,$$

we can define the following equivalent definitions:

$$\mathbf{def} f = \mathbf{Y}(F) \quad \text{where } F = \lambda f.\lambda x.E.$$

The f in E is now bound to the parameter f in F , and not the original definition f . The application of f to an argument a is evaluated as follows:

$$\begin{aligned} f(a) &= (\mathbf{Y}(F))(a) \\ &= (F(\mathbf{Y}(F)))(a) \\ &= ((\lambda f.\lambda x.E)(\mathbf{Y}(F)))(a) \\ &= (\lambda x.E')(a) \end{aligned}$$

where E' is the expression after substituting $(\mathbf{Y}(F))$ for f in E . Using the operator **Y**, we can define recursive definitions in a static-binding language without violating the binding rule. In WUPLE, the fixed-point operator **Y** is written as **rec**. As an example, the following is a recursive definition of the factorial function:

```
(def rec (fac n)
  (if (= n 0) 1
      (* n (fac (- n 1))))
)
```

In the case of auxiliary expression, we can have mutually recursive local bindings. For example,

```
(let ((f x) (... g ...))      ;; (6)
      ((g y) (... f ...))
      E
)
```

Following the idea discussed above, we use the fixed-point operator `rec` to circumvent this problem. Now (6) should be written as:

```
(let rec ((f x) (... g ...))   ;; (6')
          ((g y) (... f ...))
          E
)
```

The only restriction of recursive auxiliary expression is that *the local bindings must be lambda expressions*. If a local binding is a constant expression, we can use constant function instead, e.g., we write $x = \lambda().1$ for $x = 1$.

The fixed-point operator is the only means to introduce recursive definitions without destroying the semantics of our language.

3.5 Higher Order Functions

In general, the result of a function application is not just a simple value; it may well be a function. Furthermore, the arguments in a function application can also be functions. A function which takes functions as arguments and/or returns a function as value is called a higher order function. For example, the following function `twice` takes a unary function as an argument and returns a function which applies to its argument twice.

```
(def twice (lambda (f) (lambda (x) (f (f x)))))
```

This function `twice`, when applies to the function `succ` [cf. definition (4)], returns a function which increments its argument by 2. A function which we encounter very often in mathematics is the composition function. It can be defined as:

```
(def comp (lambda (f) (lambda (g) (lambda (x) (f (g x)))))
```

Assume that we have both `twice` and `comp` functions in our program database. We can perform the following function applications:

```
[0](def (square x) (* x x))
square
[0]->((twice square) 2)
16
[0]->((comp (twice square) square) 2)
256
```

3.6 Delayed Evaluation

When computing with infinite sequences, we often want to evaluate the elements one at a time rather than the whole sequence at once. This can be achieved only if we can explicitly delay the evaluation of the rest of the sequence once we obtain enough elements for processing. Lazy evaluation is a technique of delaying the evaluation of an expression and forcing its evaluation only when its value is needed. Two primitive functions, `delay` and `force`, are defined for this purpose, with these two primitives, we can now deal with computing infinite sequences. For instance, we can define the sequence of all integers starting from `n`:

```
(def rec (integers n) (cons n (delay (integers (+ n 1)))))
```

where `cons` is a binary function which constructs a pair out of its arguments. To get at the first element of the sequence of `integers` starting from 1, we apply the `car` function to it:

```
[0]->(car (integers 1))           ;; first element
1
```

To obtain the second element, we must explicitly force its evaluation from the given sequence.

```
[0]->(car (force (cdr (integers 1)))) ;; second element
2
[0]->(car (force (cdr (force (cdr (integers 1)))))) ;; third element
3
```

After the evaluation of the second element, the evaluation of the rest of the sequence is then delayed again.

4 System Directives and Primitives

In this section, we give a full account of all the system directives and primitives supported in WUPLE. Directives are different from primitives in the sense that they cannot be used as functions. They are designed to assist the user to interact with the WUPLE environment. Primitives are either constants or functions. They can be used in constructing new functions. In general, primitives can be treated as user functions and are evaluated by the system. As far as the user is concerned, they behave the same as any other user-defined functions. In the following subsections, we first describe the WUPLE environment features, and then all system primitive constants and functions.

4.1 Directives

There are only four system directives: `def`, `macro`, `undef` and `error`. The directive `def` and `undef` allow one to add and delete definitions into and from the program database respectively. The `macro` directive allows one to define a template and a replacement which will be used for inline text substitution on every input expression. The `error` directive is used to find out the meaning of error codes generated by the system.

`(def [rec] named-expression expression)` This directive adds a new definition, or rather a binding tuple (name,value), into the program database. The name of the new definition is the first symbol appearing in the *named-expression*; the value is a lambda expression of the form `(lambda parameters expression)` if the *named-expression* has parameters, otherwise it is just the *expression*. e.g.,

```
(def (f n) (+ n 1))
```

A binding tuple `(f,(lambda (n) (+ n 1)))` is added into the program database. e.g.,

```
(def x 1)
```

A binding tuple `(x,1)` is added into the program database. If the definition is recursive, the value of the definition is a recursive *auxiliary-expression* with the name of the definition as the body and `(named-expression,expression)` as a local binding. e.g.,

```
(def rec f <expression>)
```

A binding tuple `(f, (let rec (f <expression>) f))` is added. e.g.,

```
(def rec (f n) <expression>)
```

A binding tuple `(f, (let rec ((f n) <expression>) f))` is added. If the new definition is successfully compiled, it is added and its name is printed. Any old definition with the same name is automatically removed.

(macro *named-expression* *expression*) This directive adds a new macro definition into the system macro database. Any future input expression that matches the *named-expression* is replaced by the *expression*. Every parameter appearing in the *named-expression* is used as a place-holder for substitution in the *expression*. e.g.,

```
(macro (cons-stream x y) (cons x (delay y)))
```

The input expression `(cons-stream a b)` is replaced by the expression `(cons a (delay b))`. e.g.,

```
(macro NIL nil)
```

The input expression `(= x NIL)` is replaced by the expression `(= x nil)`. The macro expansion is always done before compilation and evaluation. Macros can be used to avoid premature evaluation of certain expressions, e.g., `cons-stream`. If a new macro definition is successfully compiled, it is added and its name is printed. Any old macro definition with the same name is removed.

(undef *symbol*) This directive is used to remove the function definition with name *symbol* from the program database permanently. It returns `true` if the definition exists, otherwise it returns `false`.

(error *integer*) When an error condition occurs, an error code is printed. This directive is used to find out what the error is and how it happens.

4.2 Primitive Constants and Functions

4.2.1 Constants

'() This constant represents the empty list. `nil` is equivalent to `'()`. `NIL` is a macro of `nil`.

false This constant represents the boolean value false. `FALSE` is a macro of `false`.

true This constant represents the boolean value true. `TRUE` is a macro of `true`.

(quote *object*) `quote` returns *object* unevaluated. It is used to denote symbolic constant. `'object` is an abbreviation of `(quote object)`. e.g.,

```
(quote (a b))   is (a b)
(quote (+ 2 3)) is (+ 2 3)
'+ 2 3         is (+ 2 3)
'a            is a
```


4.2.2 Arithmetical Functions

`(neg number)` This function returns the negative of the *number*. e.g.,

`(neg 2) is -2`

`(abs number)` This function returns the absolute value of *number*. e.g.,

`(abs -3.0) is 3.0`

`(add $n_1 n_2 \dots n_k$)` This function computes the result of $n_1 + n_2 + \dots + n_k$, $k \geq 2$, where n_1, \dots, n_k are numbers. It returns an integer if all arguments are integers. “+” is a macro of `add`. e.g.,

`(add 1 2 3 4 5) is 15`

`(sub $n_1 n_2 \dots n_k$)` This function computes the result of $n_1 - n_2 - \dots - n_k$, $k \geq 2$, where n_1, \dots, n_k are numbers. It returns an integer if all arguments are integers. “-” is a macro of `sub`. e.g.,

`(sub 10 5 3) is 2`

`(mul $n_1 n_2 \dots n_k$)` This function computes the result of $n_1 \times n_2 \times \dots \times n_k$, $k \geq 2$, where n_1, \dots, n_k are numbers. It returns an integer if all arguments are integers. “*” is a macro of `mul`. e.g.,

`(mul 1 2 3 4 5) is 120`

`(div $n_1 n_2 \dots n_k$)` This function computes the result of $n_1 \div n_2 \div \dots \div n_k$, $k \geq 2$, where n_1, \dots, n_k are numbers. It returns an integer if the result can be an integer, otherwise it returns a float. “/” is a macro of `div`. e.g.,

`(div 63 7 3) is 3`

`(quotient $n_1 n_2$)` This function computes the integer part of the quotient of $n_1 \div n_2$. The arguments must be integers. e.g.,

`(quotient 11 3) is 3`

`(mod $n_1 n_2$)` This function computes the result of $n_1 \bmod n_2$. The arguments must be integers and the result is also an integer. e.g.,

(mod -17 3) is 2

(rem n_1 n_2) This function computes the remainder of $n_1 \div n_2$. The arguments must be integers and the result is also an integer. **remainder** is a macro of **rem**. e.g.,

(rem -17 3) is -2

(integer-divide n_1 n_2) This function computes the pair “(quotient . remainder)” of $n_1 \div n_2$. The arguments must be integers. e.g.,

(integer-divide 11 3) is (3 . 2)

(log $number$) This function computes $\log_{10} number$. The result is a float. e.g.,

(log 1000) is 3.0

(pow n_1 n_2 ... n_k) This function computes the result of $n_1 ** n_2 ** \dots ** n_k$, $k \geq 2$, where n_1, \dots, n_k are numbers. It returns a float. “**expt**” is a macro of **pow**. e.g.,

(pow 2 2 2 2) is 65536.0
 (pow 2 (pow 2 (pow 2 2))) is 65536.0
 (pow (pow (pow 2 2) 2) 2) is 256.0

(ln $number$) This function computes $\log_e number$. The result is a float. e.g.,

(ln 10) is 2.3025

(exp $number$) This function computes e^{number} . The result is a float. e.g.,

(exp 1) is 2.71828

(sqrt $number$) This function computes \sqrt{number} . The result is a float. e.g.,

(sqrt 9) is 3.0

(round $number$) This function rounds off the $number$ to the nearest integer. **round** selects the integer farthest from zero. e.g.,

(round 9.5) is 10
 (round -9.5) is -10

`(trunc number)` This function returns the integer part of *number*. For positive numbers, this is the same as `floor`; for negative numbers, this is the same as `ceiling`. `truncate` is a macro of `trunc`. e.g.,

```
(trunc 9.2) is 9
(trunc -9.2) is -9
```

`(floor number)` This function returns the $\lfloor number \rfloor$, the greatest integer i such that $i \leq number$. The result is an integer. e.g.,

```
(floor 9.2) is 9
(floor -9.2) is -10
```

`(ceiling number)` This function returns the $\lceil number \rceil$, the smallest integer i such that $i \geq number$. The result is an integer. e.g.,

```
(ceiling 9.2) is 10
(ceiling -9.2) is -9
```

`(fractional-part number)` This function returns the fractional part of *number*. The result is zero if *number* is an integer. e.g.,

```
(fractional-part 9.6) is 0.6
(fractional-part 9) is 0
```

`(max n_1 n_2)` This function returns the maximum of $\{n_1, n_2\}$. The arguments must be numbers and the result is also a number. e.g.,

```
(max -17.0 30) is 30
```

`(min n_1 n_2)` This function returns the minimum of $\{n_1, n_2\}$. The arguments must be numbers and the result is also a number. e.g.,

```
(min -17.0 30) is -17.0
```

4.2.3 Trigonometrical Functions

`(acos number)` This function computes the arc cosine of *number*. The input *number* must be between 1 and -1. The result is a float in radians. e.g.,

```
(acos 0.7071) is 0.7854
```

`(asin number)` This function computes the arc sine of *number*. The input *number* must be between 1 and -1. The result is a float in radians. e.g.,

`(asin 0.7071) is 0.7854`

`(atan number)` This function computes the arc tangent of *number*. The input *number* must be any finitely representable number. The result is a float in radians. e.g.,

`(atan 1) is 0.7854`

`(cos number)` This function computes the cosine of *number*. The input *number* must be in radians. The result is a float. e.g.,

`(cos (rad 45)) is 0.7071`

`(sin number)` This function computes the sine of *number*. The input *number* must be in radians. The result is a float. e.g.,

`(sin (rad 45)) is 0.7071`

`(tan number)` This function computes the tangent of *number*. The input *number* must be in radians. The result is a float. e.g.,

`(tan (rad 45)) is 0.7071`

`(rad number)` This function computes the radian of *number* in degrees. The result is a float. e.g.,

`(rad 45) is 0.7854`

`(deg number)` This function computes the degree of *number* in radians. The result is a float. e.g.,

`(deg 1) is 57.2958`

4.2.4 Predicates

`(atom? x)` This function returns **true** if x is either a number or a symbol, otherwise it returns **false**. e.g.,

```
(atom? 1)      is true
(atom? 'a)     is true
(atom? '(a . b)) is false
```

`(float? x)` This function returns **true** if x is a float, otherwise it returns **false**. e.g.,

```
(float? 1)     is false
(float? 1.2)   is true
```

`(number? x)` This function returns **true** if x is either an integer or a float, otherwise it returns **false**. e.g.,

```
(number? 1)    is true
(number? 1.2)  is true
(number? 'a)   is false
```

`(null? x)` This function returns **true** if x is the empty list constant `'()`, otherwise it returns **false**. e.g.,

```
(null? '(a b)) is false
```

`(list? x)` This function returns **true** if x is a list, otherwise it returns **false**. e.g.,

```
(list? '(a b)) is true
(list? '())    is true
```

`(pair? x)` This function returns **true** if x is a pair, otherwise it returns **false**. e.g.,

```
(pair? '(a . b)) is true
(pair? 'a)       is false
```

`(symbol? x)` This function returns **true** if x is a symbol, otherwise it returns **false**. e.g.,

```
(symbol? 'a) is true
(symbol? 1)  is false
```

4.2.5 Relational and Logical Functions

`(eq x_1 x_2 ... x_k)` This function returns **true** if $x_1 = x_2 = \dots = x_k$, $k \geq 2$, otherwise it returns **false**. “=” and `eq?` are macros of `eq`. e.g.,

```
(= 'a 'a)           is true
(= '(1 2) '(1 3)) is false
```

`(ne x_1 x_2)` This function returns **true** if $x_1 \neq x_2$, otherwise it returns **false**. “<>” and `ne?` are macros of `ne`. e.g.,

```
(<> 'a 'a)           is false
(ne? '(1 2) '(1 3)) is true
```

`(lt x_1 x_2 ... x_k)` This function returns **true** if $x_1 < x_2 < \dots < x_k$, $k \geq 2$, otherwise it returns **false**. “<” and `lt?` are macros of `lt`. e.g.,

```
(< 1 2 4) is true
```

`(gt x_1 x_2 ... x_k)` This function returns **true** if $x_1 > x_2 > \dots > x_k$, $k \geq 2$, otherwise it returns **false**. “>” and `gt?` are macros of `gt`. e.g.,

```
(> 4 3 5) is false
```

`(le x_1 x_2 ... x_k)` This function returns **true** if $x_1 \leq x_2 \leq \dots \leq x_k$, $k \geq 2$, otherwise it returns **false**. “<=” and `le?` are macros of `le`. e.g.,

```
(<= 1 2 2 4) is true
```

`(ge x_1 x_2 ... x_k)` This function returns **true** if $x_1 \geq x_2 \geq \dots \geq x_k$, $k \geq 2$, otherwise it returns **false**. “>=” and `ge?` are macros of `ge`. e.g.,

```
(>= 4 2 2 1) is true
```

`(not expression)` This function returns **true** if *expression* returns **false**, otherwise it returns **false**. e.g.,

```
(not (eq 'a '())) is true
```

`(and x_1 x_2 ... x_k)` This function returns **true** if x_1 and x_2 and ... and x_k , $k \geq 2$, all return **true**, otherwise it returns **false**. e.g.,

```
(and (< 1 2) (< -2 1)) is true
```

`(or x_1 x_2 ... x_k)` This function returns **false** if x_1 and x_2 and ... and x_k , $k \geq 2$, all return **false**, otherwise it returns **true**. e.g.,

```
(or (< 1 1) (< 2 3)) is true
```

4.2.6 S-expression Structure Function

(car x) This function returns the left part (or head) of x if x is a pair, otherwise it is an error. e.g.,

```
(car '(a b c))      is a
(car '((a b) c d)) is (a b)
(car '())           is error
```

(cdr x) This function returns the right part (or tail) of x if x is a pair, otherwise it is an error. e.g.,

```
(cdr '(a b c))      is (b c)
(cdr '())           is error
```

(cons $x_1 x_2 \dots x_k$) This function returns a pair $((\dots((x_1 . x_2) . x_3) \dots . x_k)$, $k \geq 2$, where x_1, \dots, x_k are S-expressions. Note if x_k is '()', then the result is a list. e.g.,

```
(cons 'a 'b 'c '()) is (a b c)
(cons '(a b) '(c d)) is ((a b) . (c d))
```

(list $x_1 x_2 \dots x_k$) This function returns a list $(x_1 x_2 \dots x_k)$, $k \geq 0$, where x_1, \dots, x_k are S-expressions. e.g.,

```
(list 'a 'b 'c) is (a b c)
(list)          is ()
```

4.2.7 Environment Functions

(dictionary) This function is used to find out the names of all the definitions currently stored in the program database. It returns a list of symbols. e.g.,

```
(dictionary) is (abs list? mapcar)
```

(edit '*symbol*) This function invokes the editor (defined by the shell variable EDITOR, default is "vi") on the given file with name *symbol*. It returns **true** if the file can be edited, otherwise it returns **false**. e.g.,

```
(edit 'myfile) is true % if 'myfile' exists, else false
```

(exit) This function is used to get out of WUPLE. This function never returns.

`(display 'symbol)` This function returns the definition of the function with name *symbol* currently stored in the program database. It returns `'()` if the definition does not exist. e.g.,

```
(display 'head) is (def (head x) (car x))
```

`(load 'symbol)` This function loads and executes all the expressions stored in the file *symbol*. It returns `true` if the file *symbol* can be opened for loading, otherwise it returns `false`. e.g.,

```
(load 'myfile) is true % if 'myfile' exists, else false
```

`(wup)` This function allows one escape to WUP temporarily.

4.2.8 Delayed Evaluation Functions

`(delay expression)` This function returns a delayed *expression*. A delayed expression must be evaluated by `force`.

`(force expression)` This function forces the evaluation of the delayed *expression* and returns its result.

4.2.9 Stream Processing Functions

`(cons-stream x y)` This function returns a stream with head *x* and tail *y*. e.g.,

```
% defining a stream with two elements
(def S (cons-stream 1 (cons-stream 2 the-empty-stream)))
```

`(head stream)` This function returns the head of the *stream*. e.g.,

```
(head S) is 1 % continue from last example
```

`(tail stream)` This function returns the tail of the *stream*. e.g.,

```
(tail S) is the stream containing a single element 2
```

`(nth-element N stream)` This function returns the *N*th element of the *stream*, $N \geq 1$. e.g.,

```
(nth-element 2 S) is 2
```

`the-empty-stream` This is a constant for defining the end of a stream. It is defined as `(macro the-empty-stream '())`.

4.2.10 Miscellaneous

`(inc number)` This function always increments the *number* by 1. e.g.,

```
(inc 4) is 5
```

`(dec number)` This function always decrements the *number* by 1. e.g.,

```
(dec 4) is 3
```

`(apply func list)` This function returns the result of applying the function denoted by the *func* to the argument *list* which must be a list. e.g.,

```
(apply add '(1 2 3)) is 6
```

`(eval expression)` This function returns the result of evaluating the *expression*. e.g.,

```
(eval '(add 1 2 3)) is 6
```

`(mapcar func list)` This function applies the function *func* on each of the elements in the list *list* and returns the result as a list. e.g.,

```
(mapcar inc '(1 2 3)) is (2 3 4)
```

`(init-random integer)` This function initialises the random number generator with a seed *integer* and returns an integer, the first random number. e.g.,

```
(init-random 999) is 837457373
```

`(next-random)` This function is used to get the next random number from the initialised random number generator. It returns an integer. (Note: It must be used together with the function `init-random`.) e.g.,

```
(next-random) is 27364947
```

`(print expression)` This function prints the value of the *expression* on the user terminal and always returns `true`.

`(echo expression)` This function prints the value of the *expression* on the user terminal and returns that value.

Part II

Implementer Manual

5 System Structure

We discuss the implementation of WUPLE in this section. We assume that the reader is familiar with Prolog, which is used as our specification language. Those interested in the details of the whole implementation should consult the Appendix A, where it contains the source listing in Prolog written for WUP.

The internal structure of WUPLE is very simple. It consists of three main components: the user interface, the compiler and the SECD machine [5] emulator. The user interface which handles all system directives and macro definitions is the driver of the whole system. The compiler translates a source expression into an expression of SECD machine instructions. The machine emulator executes the SECD machine instructions and performs the primitive functions as necessary.

The organisation of this section is as follows: Section 5.1 discusses the user interface and its parser; Section 5.2 gives a description of the macro facility, the target language and its compiler; Section 5.3 talks about the SECD machine emulator and some of the modifications on the original design of Lispkit Lisp.

5.1 User Interface

This component is the main driver of the whole system. It consists of a driver, a command (or system directive) interpreter and a parser. The driver is basically equivalent to the LISP implementation's *read-eval-print* loop. The command interpreter is the top level system directives executor. Any source expression which is not a system directive is assumed to be a function application and is passed to the evaluator (the machine emulator). The parser reads and converts source S-expressions into internal Prolog list structures. The main driver is defined by the following predicate:

```
main :- next_expression( S-exp ),
        process( S-exp ),
        main.
```

The predicate `next_expression` is the parser and the predicate `process` is the top level command interpreter. These two predicates always succeed and are deterministic. This implementation heavily relies on the tail-recursion optimisation performed by WUP so that storage never runs out for infinite recursive computations.

Since our input, S-expression, has a very simple structure, the parser is basically a modified tokeniser. The parser reads and tokenises the input and at the same time constructs the corresponding internal Prolog structure for each token. e.g., the S-expression “(a b c)” is translated into the list “[a,b,c]” in Prolog.

The top level command interpreter analyses and executes three types of input: macro definitions, function definitions and function applications. If a source expression is a macro definition, it is converted into internal form and added into the macro database; if it is a function definition, it is compiled and added into the program database; otherwise, it is a function application and is compiled and then evaluated.

For a macro definition, we have only one translation rule: replace every bound variable by a unique logical variable. Using the logical variable in Prolog, the macro expansion process becomes trivial. For example, a macro definition:

```
[macro, [cons-stream, x, y], [cons, x, [delay, y]]]
```

is translated into a Prolog clause of the form:

```
macro([cons-stream, X, Y], [cons, X, [delay, Y]])
```

where `X` and `Y` are logical variables. Every macro is represented internally as a tuple in the relation `macro(Template, Replacement)`. By calling the `macro` relation with the first argument instantiated to a source expression, the output in the second argument returns the correct replacement expression with all the necessary substitutions done automatically through logical variables.

For each types of function definition, after passing the macro expansion phase, we perform some simple transformation before compile and then store it away. The transformation rules are:

1. `[def, rec, [N|P], B] => [def, N, [let, rec, [N, [lambda, P, B]], N]]`
2. `[def, rec, N, B] => [def, N, [let, rec, [N, B], N]]`
3. `[def, [N|P], B] => [def, N, [lambda, P, B]]`
4. `[def, N, B] => [def, N, B]`

Rule (4) is used for non-recursive definitions without parameters. It performs no transformation. After the transformation phase, a definition is then compiled and asserted into the program database as a tuple in the relation `code(Name, CompiledCode)`.

For a function application, we do macro expansion on the input expression, compile it to a list of SECD machine code, and then pass the resulting compiled code to the emulator.

5.2 Compiler

Our compiler is a modified version of the Lispkit Lisp compiler. It is extended with the following features: macro expansion facility, external symbol referencing, uniform user and system functions interface. To support these features, we added a few more instructions to the basic SECD machine instruction set described in Henderson's book [4]. We also decoupled the interface between the compiler and the emulator so that they can be modified in the future quite independently. In particular, the primitive function, `add, . . . , leq, etc.`, are not part of the SECD machine instruction set; they behave just like any other user-defined functions but are evaluated magically.

As mentioned earlier, all macro definitions are compiled and stored in a macro database as Prolog clauses. The macro expansion facility is handled by the following two predicates:

```

%
% expand_macro : InputExpression -> OutputExpression
%
expand_macro( [], [] ).
expand_macro( X, X' ) :-
    atomic( X ),
    try_expand_macro( X, X' ).
expand_macro( [X|Y], Z ) :-
    expand_macro( X, X' ),
    expand_macro( Y, Y' ),
    try_expand_macro( [X'|Y'], Z ).
%
% try_expand_macro : InputText -> ReplacementText
%
try_expand_macro( X, X' ) :-
    macro( X, X' ).
try_expand_macro( X, X ).

```

The predicate `expand_macro` tries all possibilities of macro expansion on the source expression. The predicate `try_expand_macro` looks up the macro definitions and replaces the input text by a replacement text if possible.

In order to assist the reader to understand the discussions that follow, we summarise the SECD machine instruction set as described in [4].

ld (N,S)	Load the value in the Nth environment with offset S on the stack (the topmost environment is the 0th environment and the beginning offset is 0). This instruction is used to get at the binding of a symbol defined in the environments.
ldc C	Load an immediate constant C on the stack.
ldf F	Load a closure for the function F on the stack.
ap	Apply the closure on top of the stack to the arguments immediately underneath.
rtn	Return from an application. It restores the machine state just before the application.
stop	Terminate program execution.
dum	Create a dummy environment which will be used for constructing a self-referenced binding environment. (Note: for recursive definitions.)
rap	Apply the current application recursively. (Note: for recursive definitions.)
sel C1,C2	Jump to code C1 if top of stack has a value true, otherwise jump to C2. It also saves the current continuation. (Note: It is similar to a branch operation.)
join	Join the main control after branching. It restores the most recently saved continuation.
ap0	Apply the closure-like object <i>recipe</i> on the stack to the arguments immediately underneath.
lde R	Load a closure-like object <i>recipe</i> on the stack.
upd	Update the content of a <i>recipe</i> .

In Lispkit Lisp, there is no facility to handle references to external symbols. In a programming environment, the user is generally allowed to add new functions incrementally and to modify the old functions if necessary. Very often the user defines a function using some other function definitions added earlier. This requires resolving references to the constantly changing external binding environment. To support this feature, we added a new instruction **ldx** and modified slightly the compilation of a new function definition.

ldx S Load the code for the external symbol S. The code loaded will become the current function being executed.

Whenever we find a symbol S which does not exist in the current binding environments,

we check the program database for its binding; if it has a binding, then we generate the instruction **ldx S**—load external symbol **S**, otherwise we report an error “unbound variable **S**?”. The **ldx** instruction behaves like an external procedure call. During execution the instruction **ldx S** will load the compiled code for the definition of **S** and the current continuation—the instruction following the **ldx**—is saved (note: not the whole machine state is saved). Now we need a way to return to this continuation after the execution of **S** terminates. The instruction **join** defined in the original SECD machine is used to restore the continuation after the branch instruction **sel**. We overload the use of **join** also for returning from an external procedure call. Every new definition is now compiled with an extra **join** instruction added to the end. e.g.,

```
[def, f, [lambda, [x], [g, x]]]
```

is compiled into the following piece of code:

```
code( f, [ldf, [ld, [0|0], ldx, g, rtn], join] )
```

In the Lispkit Lisp implementation, the primitive functions: *add*, *sub*, *mul*, *div*, *rem*, *eq*, *leq*, *atom*, *car*, *cdr* and *cons*, are part of the SECD machine instruction set. The compiler detects such function calls and generates the corresponding instructions. This creates some difficulty when these primitive functions are used as arguments to some higher order functions, e.g., (`mapcar add '((1 2) (3 4))`). One can get around this problem by adding, for each primitive functions, a definition such as (`def (add x y) (add x y)`). This solution becomes unwieldy when the number of primitive functions increases.

Our solution is to remove all primitive function instructions from the SECD machine. This simplifies somewhat the compiler implementation. Instead, we overload the use of the instruction **ldf**. The original use of **ldf** is to load a closure of a function on the stack with the arguments to the function just below this closure. When applying the closure on the stack, the environment in the closure and the arguments together become the current environments; the code for the function contains instructions, e.g., **ld (0,1)**, to extract its arguments from the environments to the stack.

When compiling a call to a primitive function **F**, we generate the instructions **ldf [F,rtn]**. The instructions **[F,rtn]** is the code part in the closure. When the function **F** is applied, its arguments will be on top of the current environments because **F** does not contain code to extract the arguments from the environment to the stack. In the SECD machine emulator, the execution of **F** will trap to a call to the system code of **F** and the result is then pushed on top of the stack (more on this in next section). The **rtn** instruction is used to restore the state of the machine before the function application. e.g.,

```
[def, suc, [lambda, [n], [+ , n, 1]]]
```

is compiled to the following SECD machine instructions:

```
code( suc, [ldf, [ld, [0|0], ldc, 1, ldf, [add, rtn], rtn], join] )
```

One primitive function instruction **cons** cannot be removed because it is also used to construct the arguments of a function application as a list. To avoid conflict with the actual primitive function **cons**, we rename the **cons** instruction to **acon** and retain its semantics.

acon Construct a pair out of the top two entries on the stack.

Now adding a new primitive function **F** amounts to adding a new fact that **F** is primitive so the compiler will generate the correct code, and adding a new piece of system code in the emulator to act upon a call to **F**. We do not have to change any place else—the compiler or the emulator. Thus we increase the modularity and flexibility of the whole system and can treat all primitive functions as any other user-defined functions. Details are given in the Appendix A.

5.3 Emulator

The SECD machine emulator is a straightforward implementation of the state transitions when applying each instructions on the machine. The driver for the emulator is the predicate `cycle` which is defined as:

```
%
% cycle : OldState -> NewState
%
cycle( CurrentState, CurrentState ) :-
    current_instruction( CurrentState, stop ).
cycle( CurrentState, FinalState ) :-
    apply( CurrentState, NewState ),
    cycle( NewState, FinalState ).
```

where a state is defined as `state(Stack,Env,Control,Dump)`. Each of the argument in `state` is implemented as a push-down list. The predicate `current_instruction` always returns the first instruction in the push-down list `Control`. The predicate `apply` applies the current instruction on the current state and returns a new state after the transition.

In our actual implementation, we capitalise on the tail-recursion optimisation and clause-indexing of WUP; we modify the predicate `cycle` and `apply` slightly so that WUP can perform the optimisations as early as possible. The modified predicates are now defined as:

```
%
% cycle : CurrentInstr, OldState -> NewState
%
cycle( stop, CurrentState, CurrentState ).
cycle( I, CurrentState, FinalState ) :-
    apply( I, CurrentState, NewState ),
    current_instruction( NewState, I' ),
    cycle( I', NewState, FinalState ).
```

```

%
% apply : CurrentInstr, OldState -> NewState
%
% (see the source listing for details)
%
```

The `apply` predicate performs the corresponding transition for each instruction. If the current instruction is `ldx S`, the remaining of the current control list is pushed onto the `Dump` list, the code for `S` is then fetched from the program database and becomes the new `Control` list; if the current instruction is not a valid machine instruction, it is checked whether it is a primitive function call; if it is a valid primitive function call, the execution is transferred to the corresponding system routine and the result is then pushed on top of the stack; otherwise an error of “illegal instruction?” is reported.

In Lispkit Lisp, the implementation of *recipe* used for delayed evaluation is realised by the “`rplaca`” function which destructively modifies a flag in the *recipe*. To avoid destructive assignment, we represent a *recipe* by a triple `recipe(Flag,Value,Closure)`. Initially, the `Flag` in the *recipe* is uninstantiated. When the *recipe* is evaluated, the `Flag` is checked; if the `Flag` is bound, then the `Value` is the current value of the *recipe*; otherwise, the `Closure` is evaluated, its value is bound to `Value` and the `Flag` is then set to a constant.

The interface between the command interpreter and the emulator is defined by the predicate `exec` which is defined as:

```

%
% exec : Application -> Result
%
exec( Application, Result ) :-
    first_instruction( Application, I ),
    cycle( I, state([],[]),Application,[]),
    state([Result|_],_,_,_) ).
```

where `Application` is the compiled version of the source application expression. `exec` initialises the current state with `Application` as the current `Control` list and returns the `Result` which is the top entry (or first element) in the `Stack` list.

6 Conclusion

In this report, we discussed an implementation of a functional programming system based on Lispkit Lisp. We extended the original implementation to include the facilities for supporting a programming environment. The whole implementation was done in Prolog—in particular, on top of the Waterloo Unix Prolog environment. The result is an integrated environment which allows the user to program in two languages within the same system. Certainly, we haven't achieved our objective to amalgamate the two languages into one; it is a first step in that direction.

WUPLE begins as an experimental system for functional programming. Its usability has not been tested. The whole implementation took about two man-weeks. It is partly because the expressive power and the non-determinism of Prolog make the whole implementation quite straightforward. When comparing to the Lispkit Lisp implementation of Lispkit Lisp, our Prolog implementation tends to be easier to understand, maintain and modify. For instance, each state transition description is directly translatable into a Prolog clause; the implementation of macro facility becomes trivial using the concept of logical variables in Prolog; parsing using the DCG (definite clause grammar) and the different-list techniques are well-known and easy to apply. Our surprising result is the speed of execution. Although WUPLE is fully interpreted, the timing on some small programs indicates that it is quite usable. Part of the reason is that we don't need a general purpose garbage-collector in Prolog. After each application, the copy stack (similar in use to a heap) is collapsed to its initial state rather than being garbage-collected. Certainly when running a fairly large application, we still need a garbage-collector for cleaning up the garbage generated during execution.

Currently, WUPLE is still undergoing some thorough testing. In the meantime, we also investigate the possibility of compiling the same input language to some other target languages, e.g., combinators [9], equations [8], etc., so as to find a suitable formalism for amalgamating functional and logic programming. So far, we have no immediate concern to re-implement WUPLE in some system programming language for the sake of efficiency. Our future research and development will mainly concentrate on approaching our ultimate goal—Assertional Programming.

7 Acknowledgements

WUPLE was proposed as a project for the course *Functional and Logic Programming* taught by Professor Maarten van Emden. Three graduate students, Dave Rosenbluet, Richard Hurley and Ken Wellsch, were involved in the initial design phase. The result was three separate projects: two implemented in C and one in Prolog. The discussions with this group of students had been illuminating. Keitaro Yukawa gave many valuable suggestions on the syntax and semantics of the input language. Professor van Emden provided the extra stimulus to my interest in functional and logic programming, in particular, after I fixed some obscure or undocumented features in Scheme.

References

- [1] Abelson, H., Sussman, G.J., Sussman, J., *Structure and Interpretation of Computer Programs*, The M.I.T. Press, Cambridge, Massachusetts, 1985.
- [2] Burge, W.H., *Recursive Programming Techniques*, Addison-Wesley, 1975.
- [3] Cheng, M.H.M., *Design and Implementation of Waterloo Unix Prolog Environment*, dissertation, Master's thesis, University of Waterloo, Canada, 1984.
- [4] Henderson, P., *Functional Programming: Application and Implementation*, Prentice Hall International, Inc., London, 1980.
- [5] Landin, P.J., *The mechanical evaluation of expressions*, Computing Journal, Vol.6, pp.308–320, 1963–4.
- [6] McCarthy, J., *Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1*, Comm. A.C.M., Vol.3, No.4, pp.184–195, 1960.
- [7] McCarthy, J., *LISP 1.5 Programmer's Manual*, Cambridge, M.I.T., 1962.
- [8] O'Donnell, M.J., *Equational Logic as a Programming Language*, The M.I.T. Press, Cambridge, Massachusetts, 1985.
- [9] Turner, D.A., *A New Implementation Technique for Application Languages*, Software-Practice and Experience, Vol.9, pp.31–49, 1979.
- [10] van Emden, M.H., Goebel, R., *Waterloo Unix Prolog User's Manual*, manuscript, University of Waterloo, Canada, 1985.

A Source Listing

```

%=====
% Module:  main  %
%=====

%
% The infinite execution loop.
%
wuple :- main.
wuple :- wuple.

%
% The main driver
%
main :-
    mark( X ),      % mark the copy stack
    next_expression( S ),
    interpret( S ),
    release( X ),  % reinitialise the copy stack
    main.

%
% The top level command driver.
% (note: it never fails and is deterministic.)
%
interpret( S ) :-
    process( S ), !.
interpret( _ ).

%
% A top level command interpreter.
%(note: it is deterministic.)
%
process( [help] ) :- !,
    write( "(def [rec] <named-expression> <expression>)\n" ),
    write( "(dictionary)\n" ),
    write( "(edit '<filename>)\n" ),
    write( "(error <integer>)\n" ),
    write( "(exit)\n" ),
    write( "(load '<filename>)\n" ),

```

```

    write( "(display '<function>)\n" ),
    write( "(macro <named-expression> <expression>)\n" ),
    write( "(undef <function>)\n" ),
    write( "(wup)\n" ).
%
% error explanation
%
process( [error,N] ) :-
    is_int( N ), !,
    error( N ).
%
% (def rec (<symbol> <parameters>) <body>)
%
process( [def,rec,[F|P],B] ) :- !,
    expand_macro( B, B' ),
    compile_define( F, [def,rec,[F|P],B],
                    [let,rec,[F,[lambda,P,B']],F] ).
%
% (def rec <symbol> <body>)
%
process( [def,rec,F,B] ) :-
    is_atom( F ), !,
    expand_macro( B, B' ),
    compile_define( F, [def,rec,F,B], [let,rec,[F,B'],F] ).
%
% (def (<symbol> <parameters>) <body>)
%
process( [def,[F|P],B] ) :- !,
    expand_macro( B, B' ),
    compile_define( F, [def,[F|P],B], [lambda,P,B'] ).
%
% (def <symbol> <body>)
%
process( [def,F,B] ) :-
    is_atom( F ), ne( F, rec ), !,
    expand_macro( B, B' ),
    compile_define( F, [def,F,B], B' ).
%
% (undef <symbol>)
%
process( [undef,F] ) :- !,
    delete_define( F ),

```

```

    print_answer( true ).
%
% (macro <named-expression> <body>)
%
process( [macro|F] ) :- !,
    compile_macro( [macro|F] ).
%
% function applications
%
process( C ) :-
    expand_macro( C, C' ), !,
    apply_function( C' ).
%
% invalid command
%
process( C ) :-
    print_answer( [illegal,construct,in,C] ).

%
% Module:    main
%
% auxiliary predicates
%

%
% compile_define( F, D, B ) :-
%   this predicate compiles a new definition B
%   (after transformation). If the compilation
%   is successful, then the new definition is added
%   with name F, original source expression D and
%   its compiled code.
%
compile_define( F, _, _ ) :-
    primitive_func( F ), !,
    put( 7 ),    % error bell
    write( "Cannot redefine primitive function: " ),
    write( F ), nl.
compile_define( F, _, _ ) :-
    special_const( F ), !,
    put( 7 ),    % error bell
    write( "Cannot redefine primitive constant: " ),
    write( F ), nl.

```

```
compile_define( F, D, B ) :-
    compile( B, C, [join], [] ), !,
    add_define( F, D, C ),
    print_answer( F ).
compile_define( F, _, _ ) :-
    print_answer( [illegal,construct,in,F] ).

%
% apply_function( C ) :-
%   this predicate compiles the application C, executes
%   it and prints the result.
%
apply_function( C ) :-
%   print_answer( C ),
    compile( C, C', [stop], [] ), !,
%   print_answer( C' ),
    exec( C', X ),
    print_answer( X ).
apply_function( C ) :-
    print_answer( [illegal,construct,in,C] ).
```

```

%=====
% Module:  compile  %
%=====

%
% compile( E, C1, C2, N ) :-
%   compile the expression E given the environment N
%   and the tail code segment C2. The code
%   generated is prepended to C2 and returned in C1.
%   (note: it is deterministic.)
%
compile( [], L, L, _ ) :- !.
compile( I, [ldc,I|L], L, N ) :-    % number constant
    is_number( I ), !.
%
% singletons
% (N.B. check local bindings before primitive constants)
%
compile( X, [ld,I|L], L, N ) :-    % variable
    is_atom(X),
    location( X, N, I ), !.
%
% built-in constants
%
compile( nil, [ldc,[]|L], L, N ).
compile( C, [ldc,C|L], L, N ) :-    % primitive constants
    primitive_const( C ), !.        % defined in 'machine'
%
% primitive functions
%
compile( F, [ldf,[F,rtn]|L], L, _ ) :-
    is_atom( F ),
    primitive_func( F ), !.         % defined in 'machine'
compile( X, [ldx,X|L], L, N ) :-    % external variable
    is_atom(X), !.
%
% lispkit lisp special constructs
%
compile( [quote,nil], [ldc,[]|L], L, N ) :- !. % (QUOTE '())
%
% N.B. the compiled code should NOT contain the constant nil.
%
```



```

compile( [quote,E], [ldc,E'|L], L, N ) :- !,    % (QUOTE E)
    replace_nil( E, E' ).
%
% delayed evaluation
%
compile( [force,E], L1, L2, N ) :- !,          % (FORCE E)
    compile( E, L1, [ap0|L2], N ).
compile( [delay,E], [lde,L1|L2], L2, N ) :- !, % (DELAY E)
    compile( E, L1, [upd], N ).
%
% conditionals
%
% (COND <CL1>...<CLn>)
% (IF E1 E2 E3)
% (ELSIF E1 E2 E3)
% (ELSE E1)
% (TRUE E1)
%
compile( [cond,C1|Cs], L1, L2, N ) :- !,
    compile_cond( [C1|Cs], L1, L2, N ).
compile( [if,E1,E2,E3], L1, L4, N ) :- !,
    compile( E3, L3, [join], N ),
    compile( E2, L2, [join], N ),
    compile( E1, L1, [sel,L2,L3|L4], N ).
compile( [elsif,E1,E2,E3], L1, L4, N ) :- !,
    compile( E3, L3, [join], N ),
    compile( E2, L2, [join], N ),
    compile( E1, L1, [sel,L2,L3|L4], N ).
compile( [else,E1], L1, L2, N ) :- !,
    compile( E1, L1, L2, N ).
compile( [true,E1], L1, L2, N ) :- !,
    compile( E1, L1, L2, N ).
%
% lambda definitions
%
% (LAMBDA (X1 ... Xk) E)
%
compile( [lambda,E1,E2], [ldf,L2|L1], L1, N ) :- !,
    compile( E2, L2, [rtn], [E1|N] ).
%
% local definitions
%
```

```

% (LETREC (X1 E1) ... (Xk Ek) E)
% (LET (X1 E1) ... (Xk Ek) E)
%
compile( [let,rec|D], [dum|L1], L2, N ) :- !,
    def_exprs( D, Es, E ),
    exprs( Es, Args ),
    vars( Es, Ns ),
    compile( E, L3, [rtn], [Ns|N] ),
    compilis( Args, L1, [ldf,L3,rap|L2], [Ns|N] ).
compile( [let|D], L1, L2, N ) :- !,
    def_exprs( D, Es, E ),
    exprs( Es, Args ),
    vars( Es, Ns ),
    compile( E, L3, [rtn], [Ns|N] ),
    compilis( Args, L1, [ldf,L3,ap|L2], N ).

%
% function applications
%
% (F A1 ... Ak)
%
compile( [F|As], L1, L2, N ) :- !,
    compile( F, L3, [ap|L2], N ),
    compilis( As, L1, L3, N ).

%
% Module:    compile
%
% auxiliary predicates
%

%
% compile_cond( E, L1, L2, N ) :-
%   given a list of clauses in E, we compile
%   it into a list of nested if-then-else expressions.
%
compile_cond( [[else,E]], L1, L2, N ) :- !,
    compile( E, L1, L2, N ).
compile_cond( [[P,E]], L1, L3, N ) :- !,
    compile( E, L2, [join], N ),
    compile( P, L1, [sel,L2,[fault,1,stop]|L3], N ).
compile_cond( [[P,E]|Y], L1, L4, N ) :- % Y is non-empty
    compile_cond( Y, L3, [join], N ),

```

```

compile( E, L2, [join], N ),
compile( P, L1, [sel,L2,L3|L4], N ).

%
% def_exprs( L, D, E ) :-
%   L is the body of a let-expression. D is the list of
%   all definitions and E is the expression in the
%   let-expression.
%
def_exprs( [E], [], E ) :- !.
def_exprs( [D|Ds], [D|Ds'], E ) :-
    def_exprs( Ds, Ds', E ).

%
% vars( B, V ) :-
%   takes a list of bindings B of the form
%   ((V1 E1) (V2 E2) ... (Vn En)) and returns a list
%   of all the variable names in V.
%   where Vi is either of the form F or (F P).
%   ((F . P) E) = (F (lambda P E))
%
vars( [], [] ) :- !.
vars( [[F|_],_|Ds], [F|Vs] ) :- !,    % Vi = (F P)
    vars( Ds, Vs ).
vars( [[D,_|Ds], [D|Vs] ) :-          % Vi = F
    vars( Ds, Vs ).

%
% exprs( B, E ) :-
%   takes a list of bindings B of the form
%   ((V1 E1) (V2 E2) ... (Vn En)) and returns a list
%   of all the expression values in E.
%   where Vi is either of the form F or (F P)
%   ((F . P) E) = (F (lambda P E))
%
exprs( [], [] ) :- !.
exprs( [[[_|P],E]|Ds], [[lambda,P,E]|Es] ) :- !,
    exprs( Ds, Es ).
exprs( [[_,E]|Ds], [E|Es] ) :-

```

```

    exprs( Ds, Es ).

%
% compilis( Es, L1, L2, N ) :-
%   this predicate compiles a list of expressions,
%   argument list, with given environment N and
%   initial code segment L2, and returns the code in L1.
%
compilis( [], [ldc,[],L], L, _ ) :- !.
compilis( [E|Es], L1, L2, N ) :-
    compile( E, L3, [acon|L2], N ),
    compilis( Es, L1, L3, N ).

%
% location( X, N, P ) :-
%   P is a location pair (E,0) of X in the namelist N,
%   where E is the env. number and 0 is the offset
%   within the env. E.
%   e.g., N has the form = [ [a,b], [x,y] ].
%   location( b, N, [0|1] ).
%   location( x, N, [1|0] ).
%
location( X, [E1|Ns], [0|0] ) :-
    member( X, E1 ), !,
    position( X, E1, 0 ).
location( X, [_|Ns], [E|0] ) :-
    location( X, Ns, [E1|0] ),
    add( 1, E1, E ).

%
% position( X, L ) :-
%   the position of X in L starting from 0.
%
position( X, [X|_], 0 ) :- !.
position( X, [_|E], N ) :-
    position( X, E, N' ),
    add( 1, N', N ).

%

```

```

% Module:  compile
%
% macro expansions
%

%
% compile_macro( M ) :-
%   this predicate compiles the macro definition M.
%   If the compilation is successful, it is added
%   into the macro database; otherwise it prints an
%   error message.
%   (note: it always succeeds and is deterministic.)
%
compile_macro( [macro,[F|P],B] ) :-
    expand_macro( B, B' ),
    generate_free_var( P, P', Vs ),
    replace_var( B', B'', Vs ),
    add_macro( [F|P'], B'' ),
    print_answer( F ).
compile_macro( [macro,F,B] ) :-
    is_atom( F ),
    expand_macro( B, B' ),
    add_macro( F, B' ),
    print_answer( F ).
compile_macro( F ) :-
    print_answer( [illegal,macro,"?",F] ).

%
% generate_free_var( P, X, V ) :-
%   Given a list of bound variables P, this predicate
%   generates a list of unique logical variables for
%   each bound variables in X and returns a list of
%   bindings in V as pairs of (Name,Variable).
%
generate_free_var( [], [], [] ) :- !.
generate_free_var( [P|Ps], [X|Xs], [[P|X]|Vs] ) :- !,
    generate_free_var( Ps, Xs, Vs ).
generate_free_var( P, X, [[P|X]] ).

%

```

```
% replace_var( P, T, V ) :-
%   this predicate replaces every bound variables in P
%   by the corresponding logical variable in V and
%   returns the resulting expression in T after replacement.
%   (note: it is deterministic.)
%
replace_var( [], [], _ ) :- !.
replace_var( [B|Bs], [X|Xs], Vs ) :-
    is_atom( B ),
    select_var( B, Vs, X ), !,
    replace_var( Bs, Xs, Vs ).
replace_var( [B|Bs], [B|Xs], Vs ) :-
    atomic( B ), !,
    replace_var( Bs, Xs, Vs ).
replace_var( [B|Bs], [X|Xs], Vs ) :-
    is_list( B ), !,
    replace_var( B, X, Vs ),
    replace_var( Bs, Xs, Vs ).
replace_var( B, X, Vs ) :-
    is_atom( B ),
    select_var( B, Vs, X ).

%
% add_macro( M ) :-
%   this predicate adds a new macro into the macro database.
%
add_macro( F, B ) :-
    delete_macro( F ),
    assert( compile, macro(F,B), [] ).

%
% delete_macro( M ) :-
%   this predicate deletes an old macro from the macro database.
%
delete_macro( F ) :-
    retract( compile, macro(F,_ ) ), !.
delete_macro( _ ).

%
```

```
% select_var( X, Vs, V ) :-
%   this predicate returns the binding of the bound variable X,
%   a logical variable, in V given the binding list Vs.
%
select_var( X, [[X|V]|_], V ) :- !.
select_var( X, [_|Vs], V ) :-
    select_var( X, Vs, V ).

%
% expand_macro( P, S ) :-
%   this predicate tries all possibilities of macro expansion
%   on the source expression P and returns the result in S.
%   (note: it is deterministic.)
%
expand_macro( [], [] ) :- !.
expand_macro( X, X' ) :-
    atomic( X ), !,
    try_expand_macro( X, X' ).
expand_macro( [X|Y], Z ) :-
    expand_macro( Y, Y' ),
    expand_macro( X, X' ),
    try_expand_macro( [X'|Y'], Z ).

%
% try_expand_macro( X, Y ) :-
%   this predicate retrieves a macro definition that
%   matches X and returns the replacement in Y. If
%   there is no suitable macro, Y is the same of the
%   source expression.
%
try_expand_macro( X, X' ) :-
    macro( X, X' ), !.
try_expand_macro( X, X ).

replace_nil( nil, [] ) :- !.
replace_nil( [E|Es], [E'|Es'] ) :- !,
    replace_nil( E, E' ),
    replace_nil( Es, Es' ).
replace_nil( [], [] ) :- !.
```

```
replace_nil( X, X ) :-  
    atomic( X ).
```



```

%=====
% Module:   machine   %
%=====

%
% The SECD machine emulator or the evaluator.
%
% exec( F, R ) :- it executes the function F on the
%   SECD machine and returns the result in R.
%
exec( [I|B], X ) :-
    cycle( I, s( ['\007'], [], [_|B], [] ), s( [X|_], _, _, _ ) ).

%
% cycle( I, S, S' ) :-
%   executes the current instruction I on the machine state S
%   and returns the final state S'.
%   (note: it is deterministic.)
%
cycle( stop, S, S ) :- !.
cycle( I, s(S,E,C,D), s(S'',E'',C'',D'') ) :-
    apply( I, s(S,E,C,D), s(S',E',[I|C'],D') ), !,
    cycle( I', s(S',E',[I|C'],D'), s(S'',E'',C'',D'') ).

%
% apply( I, S1, S2 )
%   - apply a single instruction from the SECD machine state
%     S1 to S2 with the current instruction I.
%   where S1, S2 are of form s(S,E,C,D).
%   (note: it is deterministic.)
%
apply( ld, s( S, E, [_|I|C], D ), s( [X|S], E, C, D ) ) :-
    locate( I, E, X ).
apply( ldc, s( S, E, [_|I|C], D ), s( [I|S], E, C, D ) ).
%
% save the continuation on the dump, since external code always
% contains a 'cont' instruction at the end.
%
apply( ldx, s( S, E, [_|X|C], D ), s( S, E, B, [C|D] ) ) :-
    code( X, B ), !.

```

```

apply( sel, s( [X|S], E, [_|Ct,Cf|C], D ),
        s( S, E, Cx, [C|D] ) ) :-
    select( X, Ct, Cf, Cx ).
apply( join, s( S, E, _, [C|D] ), s( S, E, C, D ) ).
apply( ldf, s( S, E, [_|C'|C], D ), s( [[C'|E]|S], E, C, D ) ).
apply( ap, s( [[C'|E']|V|S], E, [_|C], D ),
        s( [], [V|E'], C', [S,E,C|D] ) ).
apply( rtn, s( [X], _, _, [S,E,C|D] ), s( [X|S], E, C, D ) ).
%
% for recursive definitions
%
apply( dum, s( S, E, [_|C], D ), s( S, [X|E], C, D ) ).
apply( rap, s( [[C'|[V|E]]|V|S], [V|E], [_|C], D ),
        s( [], [V|E], C', [S,E,C|D] ) ).
%
% The following instructions are for 'Delayed evaluation'
%
% r( Flag, Value, Closure ) = recipe, where Flag = (t,X)
%   to indicate that the recipe has (or has not)
%   been evaluated. If Flag is free (X), then we evaluate
%   the closure, otherwise its value is in Value.
%
apply( lde, s( S, E, [_|C|C'], D ),
        s( [r(X,Y,[C|E])|S], E, C', D ) ).
apply( ap0, s( [r(X,Y,[C|E])|S], E', [_|C'], D ),
        s( [], E, C, [[r(X,Y,[C|E])|S],E',C'|D] ) ) :-
    is_var(X).
apply( ap0, s( [r(X,V,_)|S], E, [_|C], D ),
        s( [V|S], E, C, D ) ) :-
    is_atom(X), eq( X, t ).
apply( upd, s( [X], E, [_], [[r(t,X,_)|S],E',C'|D] ),
        s( [X|S], E', C', D ) ).
%
% program termination
%
apply( stop, s( S, E, _, D ), s( S, E, [stop], D ) ).
%
% exceptions
%
apply( fault, s( S, E, [_|I|C], D ), s( [M|S], E, C, D ) ) :-
    fault( I, M ).
%

```

```

% arguments construction
%
apply( acon, s( [A,B|S], E, [_|C], D ), s( [[A|B]|S], E, C, D ) ).
%
% function application (special primitive function)
%
apply( apply, s(S,[[F|E'],Arg]|E],[_|C],D),
      s([],[Arg|E'],F,[S,E,C|D]) ).
%
% system primitives
%
apply( list, s( S, [A|E], [_|C], D ), s( [A|S], E, C, D ) ).
apply( F, s( S, E, [_|C], D ), s( [A|S], E, C, D ) ) :-
    primitive0( F ), apply0( F, A ).
apply( F1, s( S, [[A]|E], [_|C], D ), s( [R|S], E, C, D ) ) :-
    primitive1( F1 ), apply1( F1, A, R ).
apply( F2, s( S, [[X,Y|Z]|E], [_|C], D ), s( [R|S], E, C, D ) ) :-
    primitive2( F2 ), apply2( F2, [X,Y|Z], R ).
%
% error conditions
%
apply( ldx, s(S,E,[_|C|_|],D), s(['\007'|S],E,[stop],D) ) :-
    write( "Unbound variable? " ), write( C ).
apply( F, s(S,E,C,D), s(['\007'|S],E,[stop],D) ) :-
    is_int( F ),
    write( "Illegal instruction ?" ).
apply( F, s(S,[A|E],C,D), s(['\007'|S],E,[stop],D) ) :-
    is_atom( F ),
    unmatched_arg_number( F, A ),
    write( "Illegal number of arguments in: " ),
    print_s_expression( [F|A] ).
apply( F, s(S,E,C,D), s(['\007'|S],E,[stop],D) ) :-
    is_atom( F ),
    write( "Exception in: " ), write( F ).

%
% auxiliary predicates or the runtime system
%

%
% nullary builtin functions.
%
```

```

apply0( next_random, X ) :- random( X ).
apply0( dictionary, X ) :- all_of( machine, X, F, define(F,_) ).
apply0( wup, true ) :-
    print_answer([execute,the,goal,"?wuple",to,return]),
    abort.
apply0( exit, true ) :- bye.
apply0( read, S ) :-
    prompt_off,
    next_expression( S ),
    prompt_on.
apply0( read, [] ).
apply0( _, false ).

%
% unary builtin functions
%
apply1( neg, X, Y ) :- is_number( X ), add( X, Y, 0 ).
apply1( car, [X|_], X ).
apply1( cdr, [_|X], X ).
apply1( not, X, Y ) :- my_not( X, Y ).
apply1( print, X, true ) :- print_answer( X ).
apply1( echo, X, X ) :- print_answer( X ).
apply1( trunc, X, Y ) :- float_to_int( X, Y ).
apply1( sin, X, Y ) :- is_number( X ), sin( X, Y ).
apply1( asin, X, Y ) :- is_number( X ), sin( Y, X ).
apply1( cos, X, Y ) :- is_number( X ), cos( X, Y ).
apply1( acos, X, Y ) :- is_number( X ), cos( Y, X ).
apply1( tan, X, Y ) :- is_number( X ), tan( X, Y ).
apply1( atan, X, Y ) :- is_number( X ), tan( Y, X ).
apply1( ln, X, Y ) :- is_number( X ), ln( X, Y ).
apply1( exp, X, Y ) :- is_number( X ), ln( Y, X ).
apply1( log, X, Y ) :- is_number( X ), log( X, Y ).
apply1( sqrt, X, Y ) :- is_number( X ), sqrt( X, Y ).
apply1( rad, X, Y ) :- is_number( X ), radian( X, Y ).
apply1( deg, X, Y ) :- is_number( X ), radian( Y, X ).
apply1( "integer?",X, Y ) :- my_integer( X, Y ).
apply1( "float?", X, Y ) :- my_float( X, Y ).
apply1( "symbol?", X, Y ) :- my_symbol( X, Y ).
apply1( init_random, X, Y ) :- is_int(X), seed(X), random(Y).
%
% system commands

```

```

%
apply1( load, File, true ) :-
    is_atom( File ),
    write( "Loading file: " ), put( '' ), write( File ),
    put( '' ), nl,
    reread( File ),
    prompt_off,
    load_file,
    close( File ),
    put( '' ), write( File ), put( '' ),
    write( " is loaded.\n" ),
    reset_prompt.
apply1( load, _, false ).
apply1( display, Name, Body ) :-
    is_atom( Name ),
    define( Name, Body ).
apply1( edit, File, true ) :-    % (edit filename)
    is_atom( File ), !,
    edit( File ).
apply1( edit, _, false ).
apply1( compile, S-exp, Code ) :-
    expand_macro( S-exp, S-exp' ),
    compile( S-exp', Code, [stop], [] ).
apply1( compile, _, [] ).
apply1( exec, Code, Result ) :-
    exec( Code, Result ).

%
% binary builtin functions
%
apply2( _, [R], R ).
apply2( add, [A,B|C], R ) :- !,
    is_number( A ), is_number( B ), add( A, B, C' ),
    apply2( add, [C'|C], R ).
apply2( cons, [A,B|C], [A|R] ) :-
    apply2( cons, [B|C], R ).
apply2( sub, [A,B|C], R ) :-
    is_number( A ), is_number( B ), add( B, C', A ),
    apply2( sub, [C'|C], R ).
apply2( mul, [A,B|C], R ) :-
    is_number( A ), is_number( B ), mul( A, B, C' ),
    apply2( mul, [C'|C], R ).

```

```
apply2( div, [A,B|C], R ) :-
    is_number( A ), is_number( B ), do_division( A, B, C' ),
    apply2( div, [C'|C], R ).
apply2( mod, [A,B], C ) :-
    is_int( A ), is_int( B ), mod( A, B, C ).
apply2( rem, [A,B], C ) :-
    is_number( A ), is_number( B ), do_division( A, B, R ),
    float_to_int( R, R1 ), mul( R1, B, R2 ), sub( A, R2, C ).
apply2( eq, [A,B], C ) :- !,
    equal( A, B, C ).
apply2( eq, [A,B|C], R ) :-
    apply2( eq, [B|C], C' ),
    apply2( eq, [A,B], B' ),
    apply2( and, [B',C'], R ).
apply2( ne, [A,B], R ) :- !,
    equal( A, B, C ),
    my_not( C, R ).
apply2( lt, [A,B], C ) :-
    atomic( A ), atomic( B ), less_than( A, B, C ).
apply2( lt, [A,B|C], R ) :-
    apply2( lt, [B|C], C' ),
    apply2( lt, [A,B], B' ),
    apply2( and, [B',C'], R ).
apply2( gt, [A,B], C ) :-
    atomic( A ), atomic( B ), less_than( B, A, C ).
apply2( gt, [A,B|C], R ) :-
    apply2( gt, [B|C], C' ),
    apply2( gt, [A,B], B' ),
    apply2( and, [B',C'], R ).
apply2( le, [A,B], C ) :-
    atomic( A ), atomic( B ), less_equal( A, B, C ).
apply2( le, [A,B|C], R ) :-
    apply2( le, [B|C], C' ),
    apply2( le, [A,B], B' ),
    apply2( and, [B',C'], R ).
apply2( ge, [A,B], C ) :-
    atomic( A ), atomic( B ), less_equal( B, A, C ).
apply2( ge, [A,B|C], R ) :-
    apply2( ge, [B|C], C' ),
    apply2( ge, [A,B], B' ),
    apply2( and, [B',C'], R ).
apply2( pow, [A,B|C], R ) :-
```

```

    is_number( A ), is_number( B ),
    apply2( pow, [B|C], C' ),
    pow( A, C', R ).
apply2( and, [A,B|C], R ) :-
    is_boolean( A ), is_boolean( B ), my_and( A, B, C' ),
    apply2( and, [C'|C], R ).
apply2( or, [A,B|C], R ) :-
    is_boolean( A ), is_boolean( B ), my_or( A, B, C' ),
    apply2( or, [C'|C], R ).

%
% supporting predicates for the runtime system
%
do_division( A, B, C ) :- mul( B, C, A ), !.
do_division( A, B, C ) :- trunc( B', B ), mul( B', C, A ).

float_to_int( X, X ) :- is_int(X), !.
float_to_int( X, Y ) :- is_float(X), trunc( X, Y ).

equal( A, A, true ) :- !.
equal( _, _, false ).

less_than( A, B, true ) :-
    lt( A, B ), !.
less_than( _, _, false ).

less_equal( A, B, true ) :-
    le( A, B ), !.
less_equal( _, _, false ).

my_atomic( [_|_], false ) :- !.
my_atomic( [], true ) :- !.
my_atomic( _, true ).

my_number( X, true ) :- is_number(X), !.
my_number( _, false ).

```

```
my_integer( X, true ) :- is_int(X), !.  
my_integer( _, false ).
```

```
my_float( X, true ) :- is_float(X), !.  
my_float( _, false ).
```

```
my_not( true, false ) :- !.  
my_not( false, true ).
```

```
is_boolean( true ) :- !.  
is_boolean( false ).
```

```
my_and( true, true, true ) :- !.  
my_and( _, _, false ).
```

```
my_or( false, false, false ) :- !.  
my_or( _, _, true ).
```

```
my_pair( [_|_], true ) :- !.  
my_pair( _, false ).
```

```
my_symbol( X, true ) :- is_atom(X), !.  
my_symbol( _, false ).
```

```
select( true, Ct, _, Ct ) :- !.  
select( false, _, Cf, Cf ).
```

```
load_file :- load_expressions.  
load_file.
```



```
load_expressions :-
    mark( X ),
    next_expression( S ),
%   print_answer( S ),
    interpret( S ),
    release( X ),
    load_expressions.

reset_prompt :-
    cur_in( stdin ), !,
    prompt_on.
reset_prompt.

add_define( F, B, C ) :-
    delete_define( F ),
    assert( machine, code( F, C ), [] ),
    assert( machine, define( F, B ), [] ).
add_define( F, B, C ) :-
    print_answer( [cannot,add,definition,F] ).

delete_define( F ) :-
    retract( machine, define( F, _ ) ),
    retract( machine, code( F, _ ) ).
delete_define( _ ).

%
% locate( I, E, V ) :-
%   returns the variable binding of I from the
%   environment E in V.
%
locate( [I|N], E, V ) :-
    index( I, E, E' ),
    index( N, E', V ).

index( 0, [S|_], S ) :- !.
index( N, [_|Ss], S ) :-
    add( 1, N', N ),
    index( N', Ss, S ).
```

```
%
% List of primitive functions
%
primitive_func( list ).      % arbitrary no. of arg.
primitive_func( F ) :- primitive0( F ).
primitive_func( F ) :- primitive1( F ).
primitive_func( F ) :- primitive2( F ).

%
% constants
%
primitive_const( nil ).
primitive_const( true ).
primitive_const( false ).

%
% nullary functions
%
primitive0( wup ).
primitive0( exit ).
primitive0( read ).
primitive0( next_random ).
primitive0( dictionary ).

%
% unary functions
%
primitive1( neg ).
primitive1( car ).
primitive1( cdr ).
primitive1( not ).
primitive1( print ).
primitive1( echo ).
primitive1( trunc ).
primitive1( sin ).
primitive1( asin ).
primitive1( cos ).
primitive1( acos ).
primitive1( tan ).
primitive1( atan ).
primitive1( ln ).
```

```
primitive1( exp ).
primitive1( log ).
primitive1( sqrt ).
primitive1( rad ).
primitive1( deg ).
primitive1( "integer?" ).
primitive1( "float?" ).
primitive1( "symbol?" ).
primitive1( init_random ).
primitive1( load ).
primitive1( display ).
primitive1( edit ).
primitive1( compile ).
primitive1( exec ).

%
% binary functions
%
primitive2( add ).
primitive2( sub ).
primitive2( mul ).
primitive2( div ).
primitive2( mod ).
primitive2( rem ).
primitive2( eq ).
primitive2( ne ).
primitive2( lt ).
primitive2( gt ).
primitive2( le ).
primitive2( ge ).
primitive2( cons ).
primitive2( pow ).
primitive2( and ).
primitive2( or ).
primitive2( apply ).

%
% exception messages
%
fault( 1, [exception,":",missing,condition,
          in,cond,expression,"?"] ).
```

```
unmatch_arg_number( F, [_|_] ) :-
    primitive0( F ).
unmatch_arg_number( F, [] ) :-
    primitive1( F ).
unmatch_arg_number( F, [_,_|_] ) :-
    primitive1( F ).
unmatch_arg_number( F, [] ) :-
    primitive2( F ).
unmatch_arg_number( F, [_] ) :-
    primitive2( F ).

special_const( X ) :- primitive_const( X ).
special_const( "NIL" ).
special_const( "TRUE" ).
special_const( "FALSE" ).
```

```

%=====%
% Module:   read   %
%=====%
%
% The parser or modified tokeniser
%

%
% next_expression( S ) :- returns the next input S-expression.
%
next_expression( S ) :-
    next_char( [], S0, 0 ),
    s_expression( S0, _, S, 0 ), !.

%
% DCG of S_expressions
% =====
%
% s_expression(['',S]) --> [quote], s_expression(S).
% s_expression([]) --> ['(',')'].
% s_expression(S) --> [C], {is_digit(C)}, number(N),
%                       {aton([C|N],S)}.
% s_expression(S) --> [C1,C2], {is_sign(C1),is_digit(C2)},
%     number(N), {aton([C2|N],S'),sign_extension(C1,S',S)}.
% s_expression(S) --> [C], {is_ident(C)}, identifier(A),
%                       {name(S,[C|A])}.
% s_expression([H|T]) --> ['('], s_expression(H),
%                       rest_expr_list(T), [')'].
%
s_expression( [C|_], S1, S, N ) :- % comment statement
    comment( C ), !,
    next_char( [], S0, N ),
    s_expression( S0, S1, S, N ).
%s_expression( ['',''|S0], S0, ["",""], _ ) :- !.
s_expression( [''|S0], S2, [quote,S], N ) :- !,
    next_char( S0, S1, N ),
    s_expression( S1, S2, S, N ).
s_expression( ['(',')'|S0], S0, nil, _ ) :- !.
s_expression( [C|S0], S1, S, _ ) :-
    is_digit( C ), !,
    number( S0, S1, A ),

```

```

    aton( [C|A], S ).
s_expression( [C1,C2|S0], S1, S, _ ) :-
    is_sign( C1 ),
    is_digit( C2 ), !,
    number( S0, S1, A ),
    aton( [C2|A], S' ),
    sign_extension( C1, S' S ).
s_expression( [C|S0], S1, S, _ ) :-
    is_ident( C ), !,
    identifier( S0, S1, A ),
    name( S, [C|A] ).
s_expression( ['(|S0], S5, [H|T], N ) :-
    add( 1, N, N1 ),          % increment the parentheses counter
    next_char( S0, S1, N1 ),
    s_expression( S1, S2, H, N1 ),
    next_char( S2, S3, N1 ),
    rest_expr_list( S3, S4, T, N1 ),
    next_char( S4, [')'|S5], N1 ).

%
% rest_expr_list(S) --> ['.'], s_expression(S).
% rest_expr_list([H|T]) --> s_expression(H), rest_expr_list(T).
% rest_expr_list([]) --> [].
%
rest_expr_list( ['.', ' '|S0], S2, S, N ) :- !,
    next_char( S0, S1, N ),
    s_expression( S1, S2, S, N ).
rest_expr_list( S0, S3, [H|T], N ) :-
    s_expression( S0, S1, H, N ), !,
    next_char( S1, S2, N ),
    rest_expr_list( S2, S3, T, N ).
rest_expr_list( S, S, [], _ ).

%
% identifier([C|Cs]) --> [C], {is_valid(C)}, identifier(Cs).
% identifier([]) --> [].
%
identifier( [C|S0], S1, [C|Cs] ) :-
    is_valid( C ), !,
    identifier( S0, S1, Cs ).

```

```

identifier( S, S, [] ).

%
% number(['.'|Cs] --> ['.'], float(Cs).
% number([C|Cs]) --> [C], {is_exponent(C)}, exponent(Cs).
% number([C|Cs] --> [C], {is_digit(C)}, number(Cs).
% number([]) --> [].
%
number( ['.'|S0], S1, ['.'|Cs] ) :- !,
    float( S0, S1, Cs ).
number( [C|S0], S1, [C|Cs] ) :-
    is_exponent( C ), !,
    exponent( S0, S1, Cs ).
number( [C|S0], S1, [C|Cs] ) :-
    is_digit( C ), !,
    number( S0, S1, Cs ).
number( S, S, [] ).

%
% float([C|Cs]) --> [C], {is_digit(C)}, float(Cs).
% float([C|Cs]) --> [C], {is_exponent(C)}, exponent(Cs).
% float([]) --> [].
%
float( [C|S0], S1, [C|Cs] ) :-
    is_digit( C ), !,
    float( S0, S1, Cs ).
float( [C|S0], S1, [C|Cs] ) :-
    is_exponent( C ), !,
    exponent( S0, S1, Cs ).
float( S, S, [] ).

%
% exponent([C|Cs]) --> [C], {is_sign(C)}, digits(Cs).
% exponent(Cs) --> digits(Cs).
%
exponent( [C|S0], S1, [C|Cs] ) :-
    is_sign( C ), !,
    digits( S0, S1, Cs ).
exponent( S0, S1, Cs ) :-

```

```
digits( S0, S1, Cs ).

%
% digits([C|Cs]) --> [C], {is_digit(C)}, digits(Cs).
% digits([]) --> [].
%
digits( [C|S0], S1, [C|Cs] ) :-
    is_digit( C ), !,
    digits( S0, S1, Cs ).
digits( S, S, [] ).

%
% auxiliary predicates
%

comment( '%' ).
comment( ';' ).

%
% valid characters in a symbol
%
is_valid( C ) :- is_alpha( C ), !.
is_valid( C ) :- is_digit( C ), !.
is_valid( C ) :- is_punct( C ).

is_digit( C ) :-
    le( '0', C ),
    le( C, '9' ).

%
% valid characters starting a symbol
%
is_ident( C ) :- is_alpha( C ), !.
is_ident( C ) :- is_punct( C ).

is_alpha( C ) :-
    le( 'a', C ),
    le( C, 'z' ).
```



```
is_alpha( C ) :-  
    le( 'A', C ),  
    le( C, 'Z' ).
```

```
is_sign( '+' ).  
is_sign( '-' ).
```

```
is_exponent( 'e' ).  
is_exponent( 'E' ).
```

```
is_punct( '=' ).  
is_punct( '?' ).  
is_punct( '!' ).  
is_punct( '"' ).  
is_punct( '@' ).  
is_punct( '$' ).  
is_punct( '#' ).  
is_punct( ':' ).  
is_punct( '-' ).  
is_punct( '_' ).  
is_punct( '<' ).  
is_punct( '>' ).  
is_punct( '&' ).  
is_punct( '^' ).  
is_punct( '|' ).  
is_punct( '+' ).  
is_punct( '*' ).  
is_punct( '/' ).  
is_punct( '.' ).  
is_punct( ',' ).  
is_punct( '[' ).  
is_punct( ']' ).  
is_punct( '{' ).  
is_punct( '}' ).  
is_punct( '%' ).  
is_punct( ';' ).
```

```
%
```

```

% tokens delimiters
%
is_special( C ) :- is_white_space( C ), !.
is_special( C ) :- reserve( C ).

is_white_space( ' ' ).
is_white_space( '\t' ).

reserve( '(' ).
reserve( ')' ).

%
% change the internal sign of a number if necessar.
%
sign_extension( '+', S, S ) :- !.
sign_extension( '-', S1, S2 ) :- add( S1, S2, 0 ).

%
% aton( L, N ) :-
%   converts a list of digits L into a number (integer or float).
%
aton( L, N ) :-
    integral_part( L, L1, 0, I ),
    fractional_part( L1, L2, 0, F ),
    add( I, F, N' ),
    exponent_part( L2, [], E ),
    mul( N', E, N ).

integral_part( [ '.' | L ], [ '.' | L ], N, N ) :- !.
integral_part( [ C | L ], [ C | L ], N, N ) :-
    is_exponent( C ), !.
integral_part( [], [], N, N ) :- !.
integral_part( [ C | L ], L1, A, V ) :-
    sub( C, '0', T ),
    mul( A, 10, A1 ),
    add( A1, T, V1 ),
    integral_part( L, L1, V1, V ).

```

```

fractional_part( [], [], F, F ) :- !.
fractional_part( [C|L], [C|L], F, F ) :-
    is_exponent( C ), !.
fractional_part( ['.'|L], L1, _, F ) :- !,
    fractional_part( L, L1, 0.0, F ).
fractional_part( [C|L], L1, _, F ) :-
    fractional_part( L, L1, 0.0, F' ),
    sub( C, '0', T ),
    div( T, 10.0, T' ),
    div( F', 10.0, F'' ),
    add( T', F'', F ).

exponent_part( [], [], 1 ) :- !.
exponent_part( [C|L], L1, E ) :-
    is_exponent( C ), !,
    exponent_part( L, L1, E ).
exponent_part( [C|L], L1, E ) :-
    is_sign( C ), !,
    exponent_part( L, L1, E' ),
    exp_sign_extension( C, E', E ).
exponent_part( L, L1, E ) :-
    integral_part( L, L1, 0, I ),
    log( E, I ).

exp_sign_extension( '+', E, E ) :- !.
exp_sign_extension( '-', E, E' ) :- div( 1.0, E, E' ).

%
% I/O handling
%
ask_input( S, N ) :-
    prompt_user( N ),
    get0( C ),
    read_line( C, S ).

read_line( '\n', [] ) :- !.
read_line( C, [C|Cs] ) :-
    get0( C0 ),
    read_line( C0, Cs ).

```

```
%
% returns the next non-white-space character
%
next_char( [], S, N ) :- !,      % end of line
    ask_input( S0, N ),
    next_char( S0, S, N ).
next_char( [C|S0], S1, N ) :-
    is_white_space( C ), !,
    next_char( S0, S1, N ).
next_char( S, S, _ ).
```

```
%=====%
% Module:   util   %
%=====%
%
% Utility predicates
%

%
% turn the system prompt on.
%
prompt_on :-
    retract( util, prompt(_) ),
    assert( util, prompt(1), [] ).

%
% turn the system prompt off.
%
prompt_off :-
    retract( util, prompt(_) ),
    assert( util, prompt(0), [] ).

%
% show the system prompt if it is on.
%
prompt_user(N) :-
    prompt(1), !,
    put( '[' ),
    write( N ),
    put( ']' ),
    write( "->" ).
prompt_user(_).

%
% print the Prolog representation of S-exp in S-exp syntax.
%
print_answer( X ) :-
    print_s_expression( X ),
    put( '\n' ).
```

```

print_s_expression( [] ) :- !,      % an empty list
    write( "()" ).
print_s_expression( [quote,X] ) :- !, % quoted expression
    put( ''' ),
    print_s_expression( X ).
print_s_expression( [X|Y] ) :-      % a pair
    atomic( Y ), !,
    put( '(' ),
    print_s_expression( X ),
    write( " ." ),
    write( Y ),
    put( ')' ).
print_s_expression( [X|Y] ) :- !,   % a list
    put( '(' ),
    print_s_exp_list( [X|Y] ),
    put( ')' ).
print_s_expression( X ) :-          % atomic elements
    write( X ).

print_s_exp_list( [] ) :- !.
print_s_exp_list( [X|Y] ) :-        % a pair
    atomic( Y ), !,
    print_s_expression( X ),
    write( " ." ),
    write( Y ).
print_s_exp_list( [X1,X2|Y] ) :-     % more one elements
    print_s_expression( X1 ),
    put( ' ' ),
    print_s_exp_list( [X2|Y] ).
print_s_exp_list( [X] ) :-           % exactly one element
    print_s_expression( X ).

```

B Sample Programs

```

;=====
; (append x y) returns the concatenation of lists x and y.
;=====
(def rec (append x y)
  (if (null? x) y
      (cons (car x) (append (cdr x) y)))
) )

```

```

;=====
; reverse(x) returns the reversal of list x.
;=====
(def (reverse x)
  (let rec
    ((rev l r) (if (null? l) r
                   (rev (cdr l) (cons (car l) r)))
    )
    (rev x NIL)
  )
) )

```

```

;=====
; (length x) returns the length of list x.
;=====
(def rec (length x)
  (if (null? x) 0
      (+ 1 (length (cdr x))))
) )

```

```

;=====
; (mapcar f l) returns a list which is the result of
;           applying the function f to each of the
;           elements in l.
;=====
(def rec (mapcar f l)
  (if (null? l) NIL
      (cons (f (car l)) (mapcar f (cdr l))))
) )

```

```

;=====
; (floor x) returns the greatest integer y such that
;           y is less than or equal to x.

```

```
=====
(def (floor x)
  (let (tx (truncate x))      ;; local binding
    (cond ((= (fractional-part x) 0) x)
          ((>= x 0) (+ 1 tx))
          (else tx))
  ) ) )
```


Index

- '() 14
- * 15
- + 15
- 15
- / 15
- < 20
- <= 20
- <> 20
- = 20
- >= 20
- > 20

- abs 15
- acos 18
- add 15
- and 20
- applicative-expression 5
- applicative order 5
- apply 23
- asin 18
- atan 18
- atom 4
- atom? 19
- auxiliary-expression 9

- body 6
- boolean 8

- car 4, 21
- cdr 4, 21
- ceiling 17
- clause 8
- combination 5
- comment 4
- cond-expression 8
- conditional-expression 8
- cons 21
- cons-stream 22
- constant-expression 8
- cos 18

- dec 23
- def 3, 13
- definition 6
- deg 18
- delay 22
- dictionary 3, 21
- display 3, 22
- div 15

- echo 23
- edit 3, 21
- eq 20
- eq? 20
- eval 23
- exit 3, 21
- exp 16
- exponent 4
- expression 7
- expt 16

- false 14
- FALSE 14
- float 4
- float? 19
- floor 17
- force 22
- fractional-part 17

- ge 20
- ge? 20
- gt 20
- gt? 20

- head 22

- if-expression 8
- inc 23
- init-random 23
- integer 4
- integer-divide 16

- lambda-expression 7

- le 20
- le? 20
- Lispkit Lisp 1
- list 4, 21
- list? 19
- ln 16
- load 3, 22
- log 16
- lt 20
- lt? 20

- macro 14
- mapcar 23
- max 17
- min 17
- mod 16
- mul 15

- named-expression 6
- ne 20
- ne? 20
- neg 15
- next-random 23
- nil 14
- NIL 14
- not 20
- nth-element 22
- null? 19
- number 4
- number? 19

- operand 5
- operator 5
- or 20

- pair 4
- pair? 19
- parameter 6
- pow 16
- print 23
- program module 2

- quote 4, 14
- quotient 15

- rad 18
- rec 3
- rem 16
- remainder 16
- round 17

- S-expression 2, 3
- Scheme 1
- sign 4
- sin 18
- sqrt 16
- sub 15
- symbol 4
- symbol? 19

- tail 22
- tan 18
- the-empty-stream 22
- true 14
- TRUE 14
- trunc 17
- truncate 17
- truth-value 8

- undef 14

- wup 22
- WUP 1
- wuple 2