

# Small Lisp Reference Manual

Robert D. Cameron

Anthony H. Dixon

Extracted from Appendix B of *Symbolic Computing with Lisp*. Prentice Hall, 1992.

## 1 Lexical Structure

A Small Lisp program consists of a stream of tokens and comment lines embedded in *white space*. White space is a sequence of one or more blanks, horizontal tabs, and newlines. Any amount of white space may separate tokens and/or comment lines, but none is generally required. The only exceptions are that some white space is required to separate consecutive atoms or identifiers (see below). White space may not appear within any token. Blanks and tab characters may appear within comment lines, however.

Small Lisp tokens can be divided into the following categories: symbolic atoms, numeric atoms, identifiers, and punctuation marks. Symbolic and numeric atoms are the elementary data objects of Small Lisp.

```
⟨symbolic-atom⟩ ::= ⟨letter⟩ { [-] ⟨letter⟩ | [-] ⟨digit⟩ } |
                  ⟨special⟩ { ⟨special⟩ }
⟨letter⟩ ::= A | B | C | ... | Z | a | b | c | ... | z
⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨special⟩ ::= + | - | * | / | < | > | = | & | | | ! | @ |
             # | $ | % | ? | :
⟨numeric-atom⟩ ::= [-] ⟨digit⟩ { ⟨digit⟩ }
```

Identifiers are used as function and variable names and have a similar syntax to that of symbolic atoms.

```
⟨identifier⟩ ::= ⟨letter⟩ { [-] ⟨letter⟩ | [-] ⟨digit⟩ }
```

The distinction between symbolic atoms and identifiers is context-dependent.

Small Lisp is case-sensitive. Thus `foo`, `Foo`, `f00`, and `F00` are distinct entities, whether interpreted as identifiers or as symbolic atoms.

The following single characters are used as punctuation marks in Small Lisp:

```
[ ] { } ( ) " = ; :
```

In addition, the three-character sequence `-->` is also a punctuation mark. Note that some of the punctuation marks are also acceptable as symbolic atoms; again, the distinction is context-dependent.

Comment lines are lines of source text beginning with three semicolons (`;;;`). One or more consecutive comment lines comprise a comment.

$$\begin{aligned} \langle \text{comment} \rangle & ::= \{ \langle \text{comment-line} \rangle \} \\ \langle \text{comment-line} \rangle & ::= \text{a line of source text beginning } ; ; ; \end{aligned}$$

Comments are descriptive text which may appear only at the level of global declarations (see Section 4.2).

## 2 Symbolic Data

The data objects of Small Lisp are called *S-expressions*.

$$\begin{aligned} \langle \text{S-expression} \rangle & ::= \langle \text{atom} \rangle \mid \langle \text{list} \rangle \\ \langle \text{atom} \rangle & ::= \langle \text{numeric-atom} \rangle \mid \langle \text{symbolic-atom} \rangle \end{aligned}$$

Atoms are the elementary data objects of the S-expression domain. Lists are structured data objects which may be composed of atoms and other lists.

$$\langle \text{list} \rangle ::= ( \{ \langle \text{S-expression} \rangle \} )$$

A list may be empty, in which case it is denoted thus:  $()$ .<sup>1</sup> Note that identifiers, and punctuation marks other than  $($  and  $)$ , do not appear in S-expressions.

The atoms **T** and **F** are used to represent the Boolean values “true” and “false,” respectively. Outside of the Boolean domain, however, **T** and **F** are simply ordinary atoms which may be used for other purposes.

## 3 Expressions

Expressions specify the computation of values in the context of sets of function definitions and variable bindings. The value of an expression is always either an S-expression or the special value  $\perp$  (“bottom”), indicating an undefined or erroneous computation. However, if  $\perp$  is the result of an evaluation, an implementation may instead report that the computation is in error with a suitable error message.

There are several kinds of expression in Small Lisp.

$$\begin{aligned} \langle \text{expression} \rangle & ::= \langle \text{value} \rangle \mid \langle \text{variable} \rangle \mid \langle \text{function-call} \rangle \mid \\ & \quad \langle \text{conditional-expression} \rangle \mid \langle \text{let-expression} \rangle \end{aligned}$$

### 3.1 Value Expressions

A value expression allows an S-expression to be represented as a value without further computation.

$$\langle \text{value} \rangle ::= \langle \text{numeric-atom} \rangle \mid " \langle \text{symbolic-atom} \rangle " \mid \langle \text{list} \rangle$$

Numeric atoms and lists can be used directly to represent themselves. Symbolic atoms must be enclosed in quotation marks to distinguish them from variable identifiers.

---

<sup>1</sup>In contrast to other Lisp dialects, the empty list is not an atom in Small Lisp.

## 3.2 Variables

Variable names use the identifier syntax.

$$\langle \text{variable} \rangle ::= \langle \text{identifier} \rangle$$

An identifier has a denotation as a variable if it is

1. One of the predefined variables **T**, **F** or **otherwise**
2. Globally defined in a constant-definition (Section 4.1)
3. Locally defined as a function parameter (Section 4.2)
4. Locally defined in a let-expression (Section 3.4).

If an identifier denotes a variable, its value is the current binding of the variable established at the point of definition. The variables **T**, **F**, and **otherwise** are predefined to have the values **T**, **F**, and **T**, respectively. If an identifier does not denote a variable, its value is  $\perp$ .

The use of an identifier as a variable name does not preclude its simultaneous use as a function name (i.e., the namespaces of variables and functions are distinct). In the context of an expression, however, an identifier is always interpreted as a variable.

## 3.3 Conditional Expressions

Conditional expressions specify alternative ways for computing a value depending on given logical conditions.

$$\begin{aligned} \langle \text{conditional-expression} \rangle & ::= [ \langle \text{clause-list} \rangle ] \\ \langle \text{clause-list} \rangle & ::= \langle \text{clause} \rangle \{ ; \langle \text{clause} \rangle \} \\ \langle \text{clause} \rangle & ::= \langle \text{expression} \rangle \text{ --> } \langle \text{expression} \rangle \end{aligned}$$

A conditional expression of  $n$  clauses each of the form  $p_i \text{ --> } r_i$  for  $1 \leq i \leq n$  evaluates to

1. The value of  $r_i$ , if for each  $k$  such that  $1 \leq k < i$ ,  $p_k$  evaluates to **F** and  $p_i$  evaluates to **T**.
2.  $\perp$ , if for each  $k$  such that  $1 \leq k < i$ ,  $p_k$  evaluates to **F** and evaluation of  $p_i$  yields a value other than **T** or **F**.
3.  $\perp$ , if for each  $k$  such that  $1 \leq k \leq n$ ,  $p_k$  evaluates to **F**.

## 3.4 Let Expressions

Let expressions allow expressions to be evaluated in the context of local variable assignments.

$$\begin{aligned} \langle \text{let-expression} \rangle & ::= \{ \langle \text{let-list} \rangle : \langle \text{expression} \rangle \} \\ \langle \text{let-list} \rangle & ::= \langle \text{local-definition} \rangle \{ ; \langle \text{local-definition} \rangle \} \\ \langle \text{local-definition} \rangle & ::= \langle \text{variable} \rangle = \langle \text{expression} \rangle \end{aligned}$$

Given an initial set of variable bindings  $E$ , a let expression having a final expression  $a$  and  $n$  local definitions each of the form  $v_i = e_i$  for  $1 \leq i \leq n$  evaluates to

1. The value of  $a$  in the context of  $E'$ , where  $E'$  is  $E$  with the additional or updated bindings of the variables  $v_i$  to the respective values of  $e_i$  each evaluated in the context of  $E$ , provided that no such evaluation yields  $\perp$ .
2.  $\perp$ , if for any  $i$ ,  $e_i$  evaluated in the context of  $E$  yields  $\perp$ .

### 3.5 Function Calls

Function calls allow user-defined or primitive functions to be called with specified argument values.

$$\begin{aligned} \langle \text{function-call} \rangle & ::= \langle \text{function-name} \rangle [ \langle \text{argument-list} \rangle ] \\ \langle \text{function-name} \rangle & ::= \langle \text{identifier} \rangle \\ \langle \text{argument-list} \rangle & ::= \langle \text{expression} \rangle \{ ; \langle \text{expression} \rangle \} \end{aligned}$$

A function call evaluates to  $\perp$  if

1. The function name specified is neither the name of a user-defined function or a primitive function.
2. The function name is that of a primitive function and the number of arguments given is not equal to the number of parameters specified in Section 5 for that primitive.
3. The function name is that of a user-defined function and the number of arguments given is not equal to the number of parameters specified in its function definition (see Section 4.2).
4. Any of the arguments evaluates to  $\perp$ .

Otherwise, the value of a function call is computed by applying the named function to the list of evaluated arguments according to the rules of Section 5 for primitive functions and Section 4.2 for user-defined functions.

## 4 Definitions and Programs

A Small Lisp program is a list of constant and function definitions.

$$\begin{aligned} \langle \text{definition-list} \rangle & ::= \{ \langle \text{definition} \rangle \} \\ \langle \text{definition} \rangle & ::= \langle \text{function-definition} \rangle \mid \langle \text{constant-definition} \rangle \end{aligned}$$

### 4.1 Constant Definitions

A constant definition establishes a variable as a globally bound constant.

$$\langle \text{constant-definition} \rangle ::= [ \langle \text{comment} \rangle ] \langle \text{variable} \rangle = \langle \text{expression} \rangle$$

The value bound to a global constant is determined by evaluating the R.H.S. of the constant definition in the context of previously defined functions and constants. It is illegal to redefine the variables **T**, **F** and **otherwise**, which have predefined values as described in Section 3.2.

## 4.2 Function Definitions

Functions may be defined by specifying a defining expression for the function which is to be evaluated in the context of argument values bound to parameter names.

$$\begin{aligned} \langle \text{function-definition} \rangle & ::= [\langle \text{comment} \rangle] \langle \text{function-name} \rangle \\ & \quad [ \langle \text{parameter-list} \rangle ] = \langle \text{expression} \rangle \\ \langle \text{parameter-list} \rangle & ::= \langle \text{variable} \rangle \{ ; \langle \text{variable} \rangle \} \end{aligned}$$

The values of the arguments are established by a function call as described in Section 3.5 and are bound to the parameter names by position (i.e., the  $i$ th argument is bound to the  $i$ th parameter). The function is applied by evaluating the defining  $\langle \text{expression} \rangle$  in the context of these bindings plus the bindings for any globally defined constants. The value computed is then returned as the result of the function application.

Function definitions may refer to any global definitions that exist at the time the function is called. This includes both self-recursive and mutually recursive function references. It is illegal to redefine any of Small Lisp's primitive functions described in Section 5.

## 5 Primitive Functions

Small Lisp provides 20 primitive functions as described below. Each function takes exactly the number of arguments shown.

`symbolp[x]` determines whether or not an object  $x$  is a symbolic atom, returning

1. T if  $x$  is a symbolic atom.
2. F if  $x$  is a numeric atom or a list.

`numberp[x]` determines whether or not an object  $x$  is a numeric atom, returning

1. T if  $x$  is a numeric atom.
2. F if  $x$  is a symbolic atom or a list.

`listp[x]` determines whether or not an object  $x$  is a list, returning

1. T if  $x$  is a list.
2. F if  $x$  is an atom.

`endp[x]` determines whether or not a list  $x$  is empty, returning

1. T if  $x$  is the empty list.
2. F if  $x$  is a nonempty list.
3.  $\perp$  if  $x$  is not a list.

`first[x]` returns

1. The first element of  $x$  if  $x$  is a nonempty list.

2.  $\perp$  if  $x$  is an empty list or an atom.

`rest[x]` returns

1. The tail sublist following the first element of  $x$  if  $x$  is a nonempty list.
2.  $\perp$  if  $x$  is an empty list or an atom.

`cons[x; y]` returns

1. The list whose first element is  $x$  and whose remaining elements are those of the list  $y$  in the same order, if  $y$  is a list.
2.  $\perp$  if  $y$  is not a list.

`eq[x; y]` determines whether  $x$  and  $y$  are equal symbolic atoms, returning

1.  $\top$  if  $x$  and  $y$  are both symbolic atoms having the same name.
2.  $\perp$  if neither  $x$  nor  $y$  is a symbolic atom.
3.  $\text{F}$ , otherwise.

`plus[x; y]` returns

1.  $x + y$  if  $x$  and  $y$  are both numeric atoms.
2.  $\perp$  if either  $x$  or  $y$  is not a numeric atom.

`minus[x; y]` returns

1.  $x - y$  if  $x$  and  $y$  are both numeric atoms.
2.  $\perp$  if either  $x$  or  $y$  is not a numeric atom.

`times[x; y]` returns

1.  $x \cdot y$  if  $x$  and  $y$  are both numeric atoms.
2.  $\perp$  if either  $x$  or  $y$  is not a numeric atom.

`divide[x; y]` computes the integer quotient of  $x$  and  $y$ , returning

1.  $\text{sgn } x \cdot \text{sgn } y \cdot \lfloor |x/y| \rfloor$  if  $x$  and  $y$  are both numeric atoms and  $y \neq 0$ .
2.  $\perp$  if  $y = 0$ , or either  $x$  or  $y$  is not a numeric atom.

`rem[x; y]` computes the integer remainder of  $x$  and  $y$ , returning

1.  $x - \text{divide}[x; y] \cdot y$  if  $x$  and  $y$  are both numeric atoms and  $y \neq 0$ .
2.  $\perp$  if  $y = 0$ , or either  $x$  or  $y$  is not a numeric atom.

`eqp[x; y]` determines whether numeric atoms  $x$  and  $y$  are equal, returning

1.  $\top$  if  $x$  and  $y$  are both numeric atoms such that  $x = y$ .

2. F if  $x$  and  $y$  are both numeric atoms such that  $x \neq y$ .
3.  $\perp$  if either  $x$  or  $y$  is not a numeric atom.

`lessp[x; y]` determines whether  $x$  is strictly less than  $y$ , returning

1. T if  $x$  and  $y$  are both numeric atoms such that  $x < y$ .
2. F if  $x$  and  $y$  are both numeric atoms such that  $x \geq y$ .
3.  $\perp$  if either  $x$  or  $y$  is not a numeric atom.

`greaterp[x; y]` determines whether  $x$  is strictly greater than  $y$ , returning

1. T if  $x$  and  $y$  are both numeric atoms such that  $x > y$ .
2. F if  $x$  and  $y$  are both numeric atoms such that  $x \leq y$ .
3.  $\perp$  if either  $x$  or  $y$  is not a numeric atom.

`sym-lessp[x; y]` determines whether  $x$  and  $y$  are lexicographically ordered symbolic atoms, returning

1. T if  $x$  and  $y$  are both symbolic atoms such that  $x$  is lexicographically less than  $y$ .
2. F if  $x$  and  $y$  are both symbolic atoms such that  $x$  is lexicographically greater than or equal to  $y$ .
3.  $\perp$  if either  $x$  or  $y$  is not a symbolic atom.

`explode[x]` returns

1. The list of single-character symbolic atoms and single-digit numeric atoms which taken together comprise the name of  $x$  if  $x$  is a symbolic atom.
2.  $\perp$  if  $x$  is a numeric atom or list.

`implode[x]` returns

1. The symbolic atom whose name is formed from the characters and digits of the atoms in the list  $x$ , if  $x$  is a list of atoms.
2.  $\perp$  if  $x$  is not a list of atoms or the characters and digits of the atoms within  $x$  do not form a valid symbolic atom name.

`error[x]` reports a run-time error, printing out  $x$  as an error message. The `error` “function” always returns  $\perp$ .

## 6 The Complete Grammar

The complete grammar of Small Lisp is collected together below.

```

    <definition-list> ::= { <definition> }
    <definition> ::= <function-definition> | <constant-definition>
    <function-definition> ::= [ <comment> ] <function-name>
        [ <parameter-list> ] = <expression>
    <parameter-list> ::= <variable> { ; <variable> }
    <constant-definition> ::= [ <comment> ] <variable> = <expression>
    <expression> ::= <value> | <variable> | <function-call> |
        <conditional-expression> | <let-expression>
    <variable> ::= <identifier>
    <value> ::= <numeric-atom> | " <symbolic-atom> " | <list>
    <function-call> ::= <function-name> [ <argument-list> ]
    <function-name> ::= <identifier>
    <argument-list> ::= <expression> { ; <expression> }
    <conditional-expression> ::= [ <clause-list> ]
    <clause-list> ::= <clause> { ; <clause> }
    <clause> ::= <expression> --> <expression>
    <let-expression> ::= { <let-list> : <expression> }
    <let-list> ::= <local-definition> { ; <local-definition> }
    <local-definition> ::= <variable> = <expression>
    <list> ::= ( { <S-expression> } )
    <S-expression> ::= <atom> | <list>
    <identifier> ::= <letter> { [-] <letter> | [-] <digit> }
    <atom> ::= <numeric-atom> | <symbolic-atom>
    <numeric-atom> ::= [-] <digit> { <digit> }
    <symbolic-atom> ::= <letter> { [-] <letter> | [-] <digit> } |
        <special> { <special> }
    <letter> ::= A | B | C | ... | Z | a | b | c | ... | z
    <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    <special> ::= + | - | * | / | < | > | = | & | | | ! | @ |
        # | $ | % | ? | :
```

## 7 Implementation Limits

A Small Lisp implementation may impose limits on the lengths of source lines, identifiers, and symbolic atom names and may also limit the range of representable numeric values.

The MSDOS implementation of Small Lisp limits the length of source lines to 254 characters. Consequently, identifiers and symbolic atom names are also limited to this length, but no further limits are imposed. The values of numeric atoms are restricted to the interval  $[-2147483648, 2147483647]$ .



## 8 Small Lisp Under MSDOS

The standard interpreter under MSDOS provides facilities for interactively evaluating Small Lisp expressions in the context of specified sets of Small Lisp definitions. The interpreter may be invoked with the command

```
smlisp [configfile]
```

The optional argument specifies an initial *configuration file* to be read in; see Section 8.4. If no configuration file is specified, the only constants and functions that are initially available are those that are predefined by the Small Lisp language. Once the interpreter has been invoked, the user may enter Small Lisp expressions to be evaluated, Small Lisp definitions to be used in subsequent evaluations, and interpreter commands.

### 8.1 Expression Entry

Small Lisp expressions may be entered directly, using as many lines as necessary. Expression input continues until the numbers of left and right parentheses, brackets, and braces are each balanced, or until input is aborted by a syntax error. The user may force an abort by typing any illegal character (e.g., a control character) followed by a return. If a function call is to be entered, note that its initial left bracket must be on the same line as the function name; otherwise it will instead be interpreted as a variable whose value is to be looked up.

### 8.2 Definition Entry

Small Lisp definitions can be entered interactively by prefixing the definition with the “@” character. As with expression entry, definitions may be entered over multiple lines, continuing until parentheses, brackets, and braces are balanced, or input is aborted by a syntax error. Alternatively, definitions may be prepared in files or configurations (see Section 8.4) and read in with the commands */read* and */reset* described in Section 8.3.

### 8.3 Interpreter Commands

Interpreter commands are entered on a single line beginning with a “/”. The following commands are available.

*/help* prints out some reminders about using the interpreter.

*/quit* stops the interpreter, returning control to the operating system.

*/read smlispfile* reads in the list of Small Lisp definitions contained in *smlispfile*, adding them to the current set.

*/reset [configfile]* clears all current definitions and optionally reads in a new configuration from *configfile*.

*/save file-name* saves all the interactively entered definitions in *file-name*.

`/show function-name` prints out the current function definition associated with *function-name*.

`/log [logfile]` closes any current log file and starts a new log on *logfile*, if specified.

## 8.4 Configuration Files

Configuration files allow sets of files containing Small Lisp definitions to be grouped together as a *configuration*. The name of each file in the configuration is placed on a separate line in the configuration file. The entire configuration can then be read with either the `smlisp` command, which starts the interpreter, or the `/reset` command described in Section 8.3.