Lips Reference Manual

Krister Joas

Version 2.0.0
January, 2023

1 Introduction

Lips is a embeddable lisp interpreter written in C++-20. In addition to a set of typical lisp functions, Lips offers functionality to make it work as a command shell. It includes functions for program composition with pipes and input/output redirection. Lips makes use of read macros and transformation of the input to make the input syntax resemble the standard shells.

Lips can be used as a lisp interpreter which can be linked with other applications. The Lips shell is an example of such an application.

Lips is inspired by Interlisp and functions are usually named as they are named in Interlisp. Functions tend to behave in the same way as they do in Interlisp. There are exceptions and Lips is not a faithful implementation of Interlisp by any stretch.

This document is a reference manual for Lips and should be adequate for getting started. See section 7 page 14 for a complete list of lisp functions provided.

When Lips is started it reads and evaluates expressions in a central initialization file. This file is called `lipsrc`, and is located in some public directory, typically `/usr/local/share/lips`. After reading this file Lips looks for the file `.lipsrc` in the users home directory. If it is found, the expressions in it are read and evaluated. Lips then reads expressions from any file given on the command line.

When the startup process is done, Lips prompts with the current history number followed by an underscore. All commands are saved on a history list, and may be recalled at any time. However, a maximum number of events saved on the history list is limited to the value of the variable `histmax`. This variable may be set by the user to any value.

2 Lips Input Syntax

The types of objects recognized by the lisp reader are integers, floats, literal atoms, lists, and strings. All objects except lists and dotted pairs are called atoms.

Strings start and end with a double quote, '`"`'. To enter a double quote in a string preceed it by a backslash, '\'. For instance, `"Foo bar"` is a string and so is `"I like \"lips\""` which contains two double quotes. Newlines may be embedded inside strings without escaping them.

Lists start with a left parenthesis followed by zero or more lisp expressions separated by separators (see below). The list ends with a matching right parenthesis. Separators breaks the atoms, and by default these are blanks, tabs, and newlines. There is also a class of characters called break characters. These characters breaks literal atoms and there is no need to add separators around them. Break characters are also literal atoms. The default break characters are: '`(`', '`)`', '`&`', '`<`', '`>`' and '`|`'. Thus,

```
foo
(foo)
(foo fie fum)
(foo (fie (x y (z))) (fum))
```

are all valid lisp expressions. The nesting of parenthesis may have arbitrary depth.

Lips supports super parenthesis. Super parenthesis are square brackets and when an opening square bracket is closed by a matching closing square bracket any missing round parenthesis is added. A final closing square bracket closes any open round parenthesis and finishes the expression. Here are some examples:

```
(cond [(null nil) "hello"] (t "world")) ⇒ "hello"
(cons 'a (cons 'b (cons 'c] ⇒ (a b c)
```

The elements of lists are stored in the car part of the cons cells, and are then linked together by the cdr's. The last cdr is always `nil`. It's possible to enter so called dotted pairs in Lips. A dotted pair is a cons cell with two expressions with no restrictions in the car and cdr of the cell. Dotted pairs are entered starting with a left parenthesis, a Lips expression, followed by a dot, '.', another expression, and terminated with a right parenthesis. `(a . b)` is an example of a dotted pair. Note that the blanks around the dot are necessary, the dot is not a break character. The dot is recognized in this special way only if it occurs as the second element from the end of a list. In other cases it is treated as an ordinary atom. The list `(. . . .)` is a list with three elements ending in a dotted pair.

A list is just a special case of dotted pairs. `(a . (b . nil))` is equivalent to `(a b)`. The second format is just a convenience since lists are so common.

Integers consist of a sequence of digits. No check for overflow is made. Floats are a bit complicated, but in most cases an atom that looks like float is a float. A float is a sequence of digits that have either a decimal point or a exponentiation character, 'e', inside. At most one decimal point is allowed. If the exponentiation character is given it must be followed by at least one digit. If these rules are not followed the atom will be interpreted as a literal atom.

Examples:

`foo`	literal atom
`"fie"`	string
`123`	integer
`12a`	literal atom
`1.0`	float
`1e5`	float
`1e`	literal atom
`.e4`	literal atom
`1.4.`	literal atom

All expressions typed at the top level prompt are treated as lists. This means that LIPS supplies an extra pair of matching parenthesis around all expressions. If the first expression of a line is an atom, and not a list, input terminates with either a return (providing that parenthesis in subexpressions match), or an extra right parenthesis. In the first case, a matching pair of parenthesis are added surrounding the line, in the second case an extra left parenthesis is added as the first character. Again, if a left parenthesis is missing a matching parenthesis is inserted.

Typing the outermost parenthesis explicitly will make LIPS print out the return value of the expression. This is the way normal lisp systems behave, but when used only as a command shell the return value is most often uninteresting. The return value is also stored in the variable *value*.

Comments are allowed in LIPS files. In order to allow for the UNIX "shebang" interpreter directive a '`#`' character in the first column of a line is recognized as a comment that ends with a newline. A `#` in any other colmn is treated as a regular character. Comments may also start with the semicolon character and the comment again continues until the end of the line.

3 The Evaluation Process

The LIPS interpreter has “dynamic scope” and “shallow” binding. “Dynamic scope” means that free variables are looked up in the call stack. For example:

```
((lambda (a) ((lambda (b) (plus a b)) 1)) 2) ⇒ 3
```

“Lexical scope” can be simulated using the `closure` function (see section **??** on page **??**) which creates a lexical scope for variables.

“Shallow” binding means that every time a symbol is bound to a value the previous value is pushed onto a stack. The current value of an atom is stored in the “value cell” and can be retrieved immediately. The opposite of “shallow” binding, “deep” binding, in contrast has to traverse the stack to find the current value of an atom.

In LIPS there is no top level value. There is also no “function definition cell”, only a “value cell”. Defining a function and binding it to a symbol simply sets the value cell for that symbol. Symbols are case sensitive and all system symbols are in lower case. LIPS does not use a stop and sweep garbage collector. Instead objects are reference counted and freed when the reference count goes down to zero.

On the top level LIPS reads expressions from standard in and evaluates the expressions. Commands entered on the top level are always treated as functions, even if no parameters are given or if the expression is a single atom. In order to print the value of a variable the function print must be used.

If the expression to be evaluated is a list the first element (the car) in the list is evaluated and then applied to the arguments. The car of the expression is reevalutated until either a proper functional form is recognized or if it’s evaluated to an illegal functional form, in which case an error is signalled.

If the functional form is an unbound atom, LIPS looks for the property LIPSautoload on the property list of the atom. If it is found, and the value is a symbol or a string, the file with that name is loaded (if possible). The interpreter then checks if the atom is no longer unbound, in wich case evaluation continues. If the atom doesn’t have the LIPSautoload property, or loading the file didn’t define the symbol, LIPS looks under the property `alias`. The atom is replaced with the expression stored under that property, if any, and the rest of the expression is appended at the end and the evaluation process continues. When all else fails it is assumed the atom stands for an executable command. The return value of an executable command is, at present, always `t`.

Note that the arguments for the command are never evaluated. If you want them evaluated you must use the function `apply`.

4 Variables

What follows is a list of all user accessable variables that in some way guides the behavior of LIPS.

`histmax` Controls the number of commands to save on the history list.

`verboseflg` If verboseflg is non-`nil`, LIPS prints some messages whenever a function is redefined or a variable is reset to a new value.

`path` The `path` variable contains a list of directories to be searched for executable programs.

`home` This variable contains the home directory of the user.

`history` In the `history` variable the history list is built. This is for internal use and proper functioning of LIPS is not garantueed if history is set to funny values.

`histnum` The variable `histnum` contains the number on the current LIPS interaction. The same warnings apply to histnum as to history.

`prompt` This variable contains the string that is printed as the prompt. An exclamation mark is replaced with the current history number from `histnum`. The default prompt is “!␣”.

`brkprompt` Same as `prompt` but controls prompting in a break. The default is “!:”.

`promptform` This variable is evaluated before the prompt is printed. If an error occurs during evaluation of promptform it is reset to the default prompt.

5 Lips as a Shell

If a function is `unbound` the Lips shell tries to run the function as an executable. When Lips reads an expression some characters are treated as read-macros. A read-macro can expand or transform the input in different ways. The current version of the reader in Lips isn't fully developed yet and it's likely to be improved in later versions.

In addition to read-macros the input is also transformed using a hook which is evaluated after an expression is read and before it's evaluated. For example, the input `ls | wc -l` is rewritten by the transform hook into `(pipe-cmd (ls) (wc -l))`. The `pipe-cmd` function evaluates each expression in a separate process, connecting the stdout from the `ls` function (i.e. the `ls` command) to the stdin of the `wc` function (command).

Redirection of stdout and stdin is also handled by the transform hook.

6 Using Lips as an Embedded Lisp Interpreter

6.1 Basic Usage

The most basic types in LIPS are the following.

- `lisp::vm` The lisp interpreter, or virtual machine. There may be only one lisp interpreter in the same program.
- `lisp::context` The context contains some globally accessable values mostly related to I/O such as the current primary output or primary error. Currently there can be only one context per program.
- `lisp_t` This is a basic type which can contain a value of several fundamental types. `lisp_t` is a pointer to a reference counted object of type `object`.

The `object` type can contain values of the following types.

`Nil` This is the value `nil` which also happens to be the `nullptr` value of `lisp_t`.

`Symbol` A literal atom.

`Integer` An integer number. This is a 64 bit integer.

`Float` A double floating point number.

`Indirect` Used in a closure.

`Cons` A cons cell consisting of two `lisp_t` values.

`String` A string.

`Subr` A compiled (C++) function. A compiled function may be spread or no spread and it may evaluate its arguments or not (fsubr). A compiled function may have zero to three arguments.

`Lambda` A lambda function. As with a compiled function a lambda may be a spread or a no spread function and it may evaluate or not evaluates its arguments (nlambda).

`Closure` A static binding.

`Environ` Internal environment stack type.

`File` A file object.

`Cvariable` A C++ variable which can be accessed and changed from both C++ and from a lisp expression.

To check the type you can use the `type_of` function which returns a class enum of type `lisp::object::type`.

If you have a value of type `object` (always indirectly via a `lisp_t` pointer) you can get the actual value using various getter functions and set the value with setter functions. There is no type checking when in the getters so calling a getter with the wrong type results in an exception.

- `auto symbol() const -> symbol::ref_symbol_t` The literal atom.
- `auto value() const -> lisp_t` Returns the value of a literal atom.
- `void value(lisp_t)` Set the value of a literal atom.
- `auto intval() const -> std::int64_t` Returns the 64 bit integer value.
- `auto floatval() const -> double` Returns the floating point value.
- `auto indirect() const -> lisp_t` Returns the real value of the indirect value.
- `auto cons() const -> const cons_t&` Returns the cons cell.
- `auto car() const -> lisp_t` Returns the `car` of the cons cell.
- `void car(lisp_t)` Sets the `car` of the cons cell.
- `auto cdr() -> lisp_t` Returns the `cdr` of the cons cell.
- `void cdr(lisp_t)` Sets the `cdr` of the cons cell.

There are some literals which simplify creating lisp objects. `operator""_s` creates a string, `operator""_a` creates a literal atom, `operator""_l` creates a number or a floating point value depending on the argument type. Finally the `operator""_e` evaluates a string as a lisp expression and returns the result of the evaluation.

```
#include lisp/lisp.hh

int main()
{
  // Create the context.
  lisp::context ctx;
  // Create the lisp interpreter.
  lisp::vm vm(ctx);
  // The _l suffix creates a lisp object by parsing
  // the string.
  lisp::print(lisp::cons("a"_a, "b"_a)); // => (a . b)
}
```

6.2 The REPL

`lisp::repl` is an object which has an `lisp_t operator()(lisp_t)` which implements a simple read, eval, and print loop. The `lisp::vm` object has a member variable called `repl` which is a function taking a `lisp_t` type object and returns a `lisp_t` type object.

Typical usage.

```
lisp::context ctx;
lisp::vm vm(ctx);
lisp::repl repl(vm);
lisp.repl = [&repl](lisp::lisp_t) -> lisp::lisp_t
{
  return repl(lisp::NIL);
};
lisp.repl(lisp::NIL);
```

If the evaluation of an expression results in a break condition, i.e. the evaluation cannot continue due to an error, then the `lisp::vm::repl` function is called recursively. The `lisp::repl` class recognizes some simple commands which allows the state to be examined or evaluation to continue, possibly after some changes to the environment or the program.

6.3 Input/Output

Input and output is handled with base classes called `source` and `sink`. A `source` needs to implement the following pure virtual functions.

`int getch()`
: Read one character from the input source.

`void ungetch(int)`
: Put a character back on the input stream to be read next time `getch` is called.

`bool close()`
: Close the input source.

`std::optional<std::string> getline()`
: Read one line from the input source. Returns an empty optional at end of file.

`iterator begin()`
: Returns an iterator which when incremented reads the next character from the source.

A `sink` needs to override the following pure virtual functions.

`void putch(int, bool)`
: Puts one character on the output stream. If the second bool parameter is `true` then characters are quoted with a backslash if non-printable.

`void puts(const std::string_view)`
: Puts a string on the output stream.

`void terpri()`
: Prints a newline.

`void flush()`
: Flushes the sink.

`bool close()`
: Closes the sink.

Several sources and sinks are predefined.

`file_source` The source is an existing file. The constructor takes a file name as its argument.

`stream_source` The stream source takes a `std::istream` as its argument.

`string_source` The string sources takes a `std::string` as its argument.

`file_sink` Takes a file name as its first argument. The second argument is a `bool` where `true` means append mode.

`stream_sink` Accepts a `std::ostream` as its argument.

`string_sink` Takes no argument. The data written to the string can be retrieved by calling the `string_sink::string()` member function.

6.4 Sharing Variables Between Lips and C++

The type `cvariable_t` is a class which enables sharing a variable of type `lisp_t` between lisp and C++.

6.5 Adding New Primitives

It's possible to create new primitive functions, functions which are registered with the lisp interpreter and are callable a lisp program. Registering a new primitive is done using the `lisp::mkprim` function. `lisp::mkprim` takes four arguments. The first is a string which is the literal atom to which the function is bound. The second argument is a C++ function which takes a lisp interpreter and zero, one, two, or three `lisp_t` values and returns a `lisp_t` value. It can be a lambda function. The third and fourth argument specifies if the function should evaluate it's argument or not and if the function is a spread or a no spread function.

Here is an example of defining a function called `printall` which takes any number of arguments and prints them.

```
mkprim(
  "printall",
  [&result](lisp\_t a) -> lisp\_t {
    for(auto p: a)
    {
      print(p);
    }
    terpri();
    return NIL;
  },
  subr_t::subr::NOEVAL, subr_t::spread::NOSPREAD);
```

7 Lisp Functions

This section describes functions available when programming in lisp. Functions are shown in their lisp forms. In `C++` the return type of each function is `LISPT`. Every function has an optional first argument which is the lisp interpreter. If left out the currently active lisp interpreter is used. For spread functions the number of arguments of type `LISPT` are the same as their lisp counterpart. For nospread functions the `C++` function has only one argument which should be a list. There is no distinction between lambda and nlambda functions in `C++` as the normal `C++` evaluation rules apply.

7.1 Arithmetic Functions

Lips supports both integer and floating point numbers. There are functions specific for either integers or floating points as well as generic functions which can take either type.

`(abs` *n*`)` [*Function*]
he absolute value of n.

`(add1` *n*`)` [*Function*]
dd `1` to n and return the value.

`(difference` *x y*`)` [*Function*]
alculates the difference between x and y.

`(difference 8 3)` ⇒ `5`

`(divide` *x y*`)` [*Function*]
ivides x by y. The result may be an integer or a floating point depending on the types of x and y. If both are integers the result will be an integer and if either x or y, or both, is a floating point number the result will be a floating point number.

`(fdifference` *x y*`)` [*Function*]
loating point difference between x and y.

`(fdivide` *x y*`)` [*Function*]
loating point division of x by y.

`(fplus` x_1 x_2 $\cdots$`)` [*NoSpread Function*]
loating point addition of x_1, x_2, …

`(ftimes` x_1 x_2 $\cdots$`)` [*NoSpread Function*]
ultiplies the floating point values x_1, x_2, …

`(idifference` *x y*`)` [*Function*]
nteger difference between x and y.

`(iminus` x`)` *[Function]*
ame as `(idifference 0 x)`.

`(iplus` x_1 x_2 $\cdots$`)` *[NoSpread Function]*
dds the values x_1, x_2, ...

`(iquotient` x y`)` *[Function]*
he quotient of the integers x and y.

`(iremainder` x y`)` *[Function]*
he integer remainder of x and y.

`(itimes` x_1 x_2 $\cdots$`)` *[NoSpread Function]*
ultiplies the integer values x_1, x_2, ...

`(itof` x`)` *[Function]*
onvert an integer x to a value of floating point type.

`(minus` x`)` *[Function]*
eturns the equivalent of `(difference 0 x)`.

`(minusp` x`)` *[Function]*
eturns `t` if x is less than zero, `nil` otherwise.

`(numberp` x`)` *[Function]*
eturns x if x is either an integer or a floating point number, `nil` otherwise.

`(plus` x_1 x_2 $\cdots$`)` *[NoSpread Function]*
um up the values x_1, x_2, ...

`(sub1` x`)` *[Function]*
eturns `(plus x 1)`.

`(times` x_1 x_2 $\cdots$`)` *[NoSpread Function]*
ultiplies the numbers x_1, x_2, ...

7.2 Variables

Functions for setting the value cell of an atom.

`(set` a e`)` *[Function]*
ets the value cell of a to the value of the expression e.

`(setq` a e`)` *[NLambda Function]*
ame as `set` but doesn't evaluate the first argument a.

`(setqq` a e`)` *[NLambda Function]*
ame as `set` but neither a nor e are evaluated.

7.3 Logic Functions

All logic functions return `nil`if the function evaluates to false and non-`nil`otherwise.

(`and` x_1 x_2 $\cdots$) *[NoSpread NLambda Function]*

valuates each argument x_1, x_2, ... in sequence and returns the value of the last expression if all arguments evaluates to non-`nil`. As soon as an argument which evaluates to `nil` is encountered, `nil` is returned. Thus the function short circuits the arguments.

(`not` x) *[Function]*

eturns `nil` if x is non-`nil` and `t` otherwise.

(`or` x_1 x_2 $\cdots$) *[NLambda Function]*

valuates each argument x_1, x_2, ... in sequence and returns `nil` if all arguments evaluates to `nil`. As soon as an argument is evaluated to a non-`nil` value that value is returned.

7.4 String Functions

(`concat` x_1 x_2 $\cdots$) *[Function]*

oncatenate the strings x_1, x_2, ...Strings are copied.

(`strcmp` s_1 s_2) *[Function]*

ompares the two strings s_1 and s_2 lexicographically and returns a negative number, zero, or a positive number if s_1 is less than, equal, or greater than s_2.

(`strequal` s_1 s_2) *[Function]*

ompares the strings s_1 and s_2 and returns `t` if the strings are equal or `nil` if they are not equal.

(`stringp` s) *[Function]*

eturns `t` if s is a string, `nil` otherwise.

(`strlen` s) *[Function]*

eturns the length of the string s.

(`substring` x n m) *[Function]*

reates a new string which is a substring of x starting from the nth character through the mth character. If m is `nil` the substring starts from the nth character through to the end.

Negative numbers are treated as the nth or mth character from the end.

```
(substring "hello" 2 3) ⇒ "el"
(substring "hello" 2 1) ⇒ nil
(substring "hello" 2 -2) ⇒ "ell"
```

```
(substring "hello" -3 -1) ⇒ "llo"
(substring "hello" -1 -3) ⇒ nil
```

`(symstr` *a*`)` [*Function*]

eturns a symbol's print string.

7.5 List Functions

`(append` x_1 x_2 $\cdots$`)` [*Function*]

ppend all the arguments, i.e. x_2 is appended to x_1, x_3 to x_2, and so on. All arguments must be lists. Any argument which is `nil` is ignored. All lists are copied which means that `(append x)` makes a copy of *x*. All arguments have to be lists.

```
(append '(a b) '(c d) '(e f)) ⇒ (a b c d e f)
```

`(attach` *x* *y*`)` [*Function*]

estructive version of `cons` which prepends *x* to *y* and any reference to *y* will contain the modified list.

`(car` *x*`)` [*Function*]

eturns the value stored in the head of the cons cell. If *x* is `nil` it returns `nil`.

`(cdr` *x*`)` [*Function*]

eturns the value stored in the tail of the cons cell. If *x* is `nil` it returns `nil`.

`(caaar` *x*`)` [*Function*]

(car (car (car x)))

`(caadr` *x*`)` [*Function*]

(car (car (cdr x)))

`(caar` *x*`)` [*Function*]

(car (car x))

`(cadar` *x*`)` [*Function*]

(car (cdr (car x)))

`(caddr` *x*`)` [*Function*]

(car (cdr (cdr x)))

`(cadr` *x*`)` [*Function*]

(car (cdr x))

`(cdaar` *x*`)` [*Function*]

(cdr (car (car x)))

`(cdadr` x`)` *[Function]*

(cdr (car (cdr x)))

`(cdar` x`)` *[Function]*

(cdr (car x))

`(cddar` x`)` *[Function]*

(cdr (cdr (car x)))

`(cdddr` x`)` *[Function]*

(cdr (cdr (cdr x)))

`(cddr` x`)` *[Function]*

(cdr (cdr x))

`(cons` a `b)` *[Function]*

reate a `cons` cell and populate the head and the tail with the values a and b. If b is left out the tail will be `nil`. If both a and b are left out then both the head and the tail will be `nil`.

```
(cons 'a 'b) ⇒ (a . b)
(const 'a '(b)) ⇒ (a b)
(cons) ⇒ (nil)
```

`(length` l`)` *[Function]*

eturns the length of the list l.

`(list` x_1 x_2 $\cdots$`)` *[Function]*

reate a list of the items x_1, x_2, ...

`(nconc` x_1 x_2 $\cdots$`)` *[Function]*

ame as `append` but modifies the arguments x_1, x_2, ...

`(nth` x n`)` *[Function]*

eturns the nth tail of x. Returns `nil` if there are fewer elements in x than n.

```
(nth '(a b c) 2) ⇒ (b c)
(nth '(a b c) 4) ⇒ nil
```

`(memb` x y`)` *[Function]*

ooks for an element x in y using `eq`, returning the tail with that element at the head. Returns `nil` if not found.

```
(memb 'a '(a b c)) ⇒ (a b c)
(memb 'b '(a b c)) ⇒ (b c)
(memb 'd '(a b c)) ⇒ nil
```

`(rplaca` x y`)` *[Function]*

eplaces `car` of x with y destructively.

`(rplacd` x y`)` *[Function]*

eplaces `cdr` of x with y destructively.

`(tconc` l o`)` *[Function]*

he `car` of l is a list and the `cdr` of l is a pointer to the first element of the list. The object o is added to the end of the list and the `cdr` is updated. An empty l should be `(nil)` but if l is `nil` it is initialized to `((o) o)`. All pointers to l points to the new list since the changes are destructive.

7.6 Functions to Function and Evaluate Functions

`(apply` fn l`)` *[Function]*

pplies the function fn to the arguments in the list l as if fn is called with the list as its arguments.

`(apply car '((a b c)))` $\Rightarrow$ `a`

`(apply* fn` x_1 x_2 $\cdots$`)` *[NoSpread Function]*

nospread version of `apply`.

`(apply* car '(a b c))` $\Rightarrow$ `a`

`(closure` f v`)` *[Function]*

closure Eval function that forms the closure of function f, with variables listed in v statically bound. This is close to function in other lisp dialects. Any closure that is created within another closure and lists a variable contained in that closure refers to the same variable. This makes it possible for two closures to share one or more variables.

Here is an example of defining a simple function which maintains the balance of a bank account.

```
(defineq
  (make-account
   (lambda (balance)
     ((closure
          '(progn
            (setq withdraw
             (closure
                 (lambda (amount)
                   (setq balance (difference
                      balance amount)))
                 '(balance)))
            (setq deposit
             (closure
                 (lambda (amount)
```

```
14                      (setq balance (plus balance
                           amount)))
15                    '(balance)))
16               (lambda ()
17                 (closure
18                     (lambda (m)
19                       (cond
20                         ((eq m 'withdraw) withdraw)
21                         ((eq m 'deposit) deposit)
22                         (t nil)))
23                     '(withdraw deposit))))
24             '(balance withdraw deposit))))))
```

The function `make-account` creates and returns a closure object which binds the three symbols on line 24 in their lexical scope. It sets the symbols `withdraw` and `deposit` each to a closure over `balance` with a lambda expression which subtracts or adds an `amount` to the `balance`.

`(define` x`)` *[Function]*

unctions functions according to x. Each element of the list x is a list of the form `(name args . body)`. The list is evaluated.

`(defineq` x $\cdots$`)` *[NLambda NoSpread Function]*

unctions functions according to the list x. Each element of the list x is of the form `(name def)` where name is the name of the function and the def is a lambda expression.

`(defineq (double (lambda (x) (times 2 x))))` ⇒ `(double)`

`(eval` e`)` *[Function]*

valuate the expression e.

`(evaltrace` n`)` *[Function]*

ets the trace variable to n. If n is greater than zero the interpreter will print some more details on how expressions are evaluated. Call `evaltrace` with a zero argument to turn off the tracing.

`(e` x`)` *[NLambda Function]*

lambda version of `eval`. Evaluates x. To illustrate the difference between the functions `eval` and `e`:

`(setq f '(plus 1 2))` ⇒ `(plus 1 2)`
`(e f)` ⇒ `(plus 1 2)`
`(eval f)` ⇒ `3`

`(getrep` x`)` *[Function]*

eturns the function definition of a `lambda` or a `nlambda` function object. Calling `getrep` on any other type of object (including `subr` and `fsubr`) returns `nil`.

```
(defineq (double (lambda (x) (times 2 x)))) ⇒ (double)
(getrep double) ⇒ (lambda (x) (times 2 x))
double ⇒ #<lambda 7fec0a810ce0>
(getrep apply) ⇒ nil
```

`(lambda` x `.` y`)` *[Function]*

reates a lambda object. The parameter x is the parameters of the function being defined. If it's a list of atoms the function is a spread function, if it's a single atoms the function is a nospread function, if it's dotted pair the function is a half spread function.

A "spread" function binds each formal parameter to the actual parameters when the function is called. Any excess parameter is ignored and any missing actual parameter is bound to `nil`.

A "nospread" function binds the formal parameter to a list of all actual parameters when called.

A "half spread" function is a combination of the above where the actual parameters are bound to each formal parameter and any excess actual parameters are bound to the formal parameter in the symbol in the `cdr` of the list of formal parameters.

`(nlambda` x `.` y`)` *[Function]*

ame as `lambda` except that the function object is an nlambda function object and parameters are not evaluated when the function is called.

`(prog1` x_1 x_2 $\cdots$`)` *[Function]*

valuate each expression in sequence, returning the result of the first expression.

`(progn` x_1 x_2 $\cdots$`)` *[Function]*

imilar to `prog1` but instead the value of the last expression is returned.

`(quote` x`)` *[NLambda Function]*

eturns x unevaluated.

7.7 Predicates

`(atom` x`)` *[Function]*

redicate which is true if x is an atom, either a symbol or a number. Strings, for example, are not atoms.

`(boundp` x`)` *[Function]*

valuates to `t` if x is not bound to a value. Note that the argument x is evaluated.

```
(boundp 'x) ⇒ t
```

if x is not bound to a value.

(eq x y) *[Function]*

eturns `t` if x is the same object as y.

(eqp x y) *[Function]*

f both x and y are numbers then `eqp` returns `t` if the numbers are the same, otherwise return the (eq x y).

(equal x y) *[Function]*

eturn `t` if x and y are `eq`, or if x and y are `eqp`, or if x and y are `strequal`, or if x and y are lists (and (equal (car x) (car x)) (equal (cdr x) (cdr y))).

(geq x y) *[Function]*

if $x \geq y$, otherwise `nil`.

(greaterp x y) *[Function]*

if $x > y$, otherwise `nil`.

(leq x y) *[Function]*

if $x \leq y$, otherwise `nil`.

(lessp x y) *[Function]*

if $x < y$, otherwise `nil`.

(listp x) *[Function]*

if x is a list, otherwise `nil`.

(litatom x) *[Function]*

if x is a literal atom, otherwise `nil`.

(neq x y) *[Function]*

if x is not `eq` to y, otherwise `nil`. Equivalent to (not (eq x y)).

(neqp x y) *[Function]*

if x is not `eqp` to y, otherwise `nil`. Equivalent to (not (eqp x y)).

(nlistp x) *[Function]*

if x is not a list, otherwise `nil`. Same as (not (listp x)).

(null x) *[Function]*

if x is `nil`, otherwise `nil`.

(symbolp x) *[Function]*

ame as (litatom x).

(zerop x) *[Function]*

if x is zero, otherwise `nil`.

7.8 Property List Functions

LIPS supports property lists on literal atoms. A property list is a list of values stored in the "property cell" of a literal atom. A property list is list of alternating properties and values. For example the property list `(a 1 b 2)` has two properties `a` and `b` with the values `1` and `2` respectively. `eq` is used to compare properties when manipulating the property list with the below functions.

`(getplist` *a*`)` [*Function*]
eturns the entire property list stored in the property cell of *a*.

`(getprop` *a p*`)` [*Function*]
eturns the value of property *p* stored in the property cell of the literal atom *a*.

`(putprop` *a p v*`)` [*Function*]
uts the value *v* in the property *p* of *a*.

`(remprop` *a p*`)` [*Function*]
emoves the property *p* from the literal atom *a*.

`(setplist` *a pl*`)` [*Function*]
ets the property list of *a* to *pl*.

7.9 Input and Output Functions

All output functions taking a file descriptor *f* as a parameter operates on the primary output if *f* is `nil`, or on primary error if *f* is `t`, otherwise *f* must be a file handle open for writing.

For input functions the file descriptor *f* is instead primary input if *f* is `nil`, stdin if *f* is `t`, or a file handle of a file open for reading.

`(close` *f*`)` [*Function*]
loses the file associated with f.

`(load` *fn*`)` [*Function*]
oads a file, evaluating each s-expression.

`(open` *fn*`)` [*Function*]
pen a file.

`(print)` [*Function*]
rint in such a way that whatever is printed can be read back by the interpreter. This means including double quotes around strings and quoting special characters with backslashes.

`(prin1` *x f*`)` [*Function*]
rints the arguments without escapes, i.e. strings are printed without surrounding double quotes, and without a terminating newline.

`(prin2` x_1 x_2 $\cdots$`)` [*Function*]

ame as `print` but without a terminating newline.

`(printlevel)` [*Function*]

pecifies how deep printing should go.

`(ratom` f`)` [*Function*]

ead an atom from f.

`(read` f`)` [*Function*]

ead an S-expression from f.

`(readc` f`)` [*Function*]

ead one character from f.

`(readline` f`)` [*Function*]

ead a line up to a newline from f.

`(spaces` n f`)` [*Function*]

rint n spaces to f.

`(terpri` f`)` [*Function*]

rint a newline to f.

7.10 Conditionals and Control Functions

`(cond (`$\mathtt{p}_1$ $\mathtt{e}_1$ $\cdots$`) (`$\mathtt{p}_2$ $\mathtt{e}_2$ $\cdots$`)` $\cdots$`)` [*Function*]

he predicates p_n are evaluated in order and the first that is evaluated to anything other than `nil`, the expressions after that predicate is evaluated. The last expression evaluated is returned as in progn. If no predicate evaluates to non-`nil`, `nil` is returned.

`(if` p t `.` f`)` [*NLambda Function*]

f the predicate p evaluates to a non-`nil` value the expression t is evaluated and returned from the function. If p evaluates to `nil` then the value of the expression f is returned.

`(while` p x_1 x_2 $\cdots$`)` [*NLambda Function*]

hile the predicate p evaluates to a non-`nil` value the sequence of expressions x_1, x_2, ...are evaluated in sequence. `while` always returns `nil`.

7.11 Map Functions

Map functions iterate over a list of items and applies a function to either each item or each tail of the list.

`(map` *list* fn_1 fn_2`)` [*Function*]

f fn_2 is `nil`, apply the function fn_1 on each tail of *list*. First fn_1 is applied to *list*, then `(cdr list)`, and so on. If fn_2 is not `nil` then fn_2 is called instead of `cdr` to get the next value on which to apply fn_1. `map` returns `nil`.

```
←(map '(a b c) (lambda (l) (print l)))
(a b c)
(b c)
(c)
nil
```

`(mapc` *list* fn_1 fn_2`)` [*Function*]

mapc is the same as `map` except that fn_1 is applied to `(car list)` instead of the list. Effectively applying fn_1 on each element of the list. `mapc` returns `nil`.

`(maplist` *list* fn_1 fn_2`)` [*Function*]

he same as `map` but collects the results from applying fn_1 on each tail and returning a list of the results.

```
(maplist '(a b c) (lambda (l) (length l))) ⇒ (3 2 1)
```

`(mapcar` *list* fn_1 fn_2`)` [*Function*]

quivalent to `mapc` but collects the results in a list like `maplist`.

```
(mapcar '(1 2 3) (lambda (n) (plus n 1))) ⇒ (2 3 4)
```

7.12 Special Functions

`(backtrace)` [*Function*]

rints a backtrace of the evaluation stack when called.

`(exit` *code*`)` [*Function*]

hrows the exception `lisp::lisp_finish` which contains the member variable `exit_code`. This member variable is set to the value of *code*. The main function of the C++ program should catch this exception and call `exit(ex.exit_code)`.

`(freecount)` [*Function*]

eturns the number of free cons cells available. Since LIPS doesn't manage memory with garbage collection this value is rather useless.

`(topofstack)` [*Function*]

eturns the current environment.

`(destblock` *e*`)` [*Function*]

eturns the destination block from an environment. Here is an example showing the destination block in some cases.

```
(setq a 88) ⇒ 88
(defineq
  (f0 (lambda (a) (destblock (topofstack))))
  (f1 (lambda (a) (f0 a)))) ⇒ (f0 f1)
(f1 99) ⇒ (1 (a .  99))
(f0 101) ⇒ (1 (a .  88))
```

`(typeof` *x*`)` [*Function*]

eturns a literal atom describing the data type of *x*.

```
(typeof 'a) ⇒ symbol
(typeof 100) ⇒ integer
(typeof nil) ⇒ nil
(typeof t) ⇒ t
(typeof 1.0) ⇒ float
(typeof "string") ⇒ string
```

`(obarray)` [*Function*]

eturns a list of all literal atoms currently active in the system.

7.13 Shell Functions

This section describes functions which implement shell features like output redirection, changing the working directory, job control, etc.

All functions that in some way redirects its input or output are executed in a fork. This means that redirecting I/O of a lisp function doesn't make permanent changes to the Lips environment. No global variables are changed.

All functions in this section are nlambda functions.

`(cd` *d*`)` [*NLambda Function*]

hanges current working directory to *d*.

`(redir-to` *cmd file fd*`)` [*NLambda Function*]

valuates the expression *cmd*, redirecting the output from this expression to a file with the file name *file*. The third parameter *fd* is optional and if given should be the file descriptor to redirect. The default is to redirect stdout.

`(redir-from` *cmd file fd*`)` [*NLambda Function*]

valuates the expression *cmd*, redirecting the file descriptor *fd* (or stdin if not given) from the file *file*.

`(append-to` *cmd file fd*`)` [*NLambda Function*]

ame as `redir-to` but append to the file *file* instead of overwriting it.

`(pipe-cmd` x_1 x_2 $\cdots$`)` [*NLambda Function*]

onnects the stdout of x_1 to stdin of x_2 and stdout of x_2 to stdin of x_3 and so on. If there is only one expression it is simply evaluated.

`(back` *cmd*`)` [*NLambda Function*]

reates a new process and evaluates the command *cmd* in this process. A new job is created which can be controlled by the job controlling functions.

`(stop)` [*NLambda Function*]

ends a stop signal to itself.

`(rehash)` [*NLambda Function*]

keeps a hash of all executable files in the `path` so that it can fail fast if a program is not available. The `rehash` function rebuilds this hash in order to pick up new programs.

`(jobs)` [*NLambda Function*]

ists all jobs currently in flight.

`(fg` *job*`)` [*NLambda Function*]

rings the job *job* running in the background to the foreground.

`(bg` *job*`)` [*NLambda Function*]

ontinues the job *job* in the background.

`(setenv` *var* *val*`)` [*NLambda Function*]

ets the environment variable *var* to the value *val*.

`(getenv` *var*`)` [*NLambda Function*]

eturns the value of the environment varable *var* or `nil` if the variable is not set.

`(exec` *cmd*`)` [*NLambda Function*]

eplaces the current process with *cmd*.

8 Using Lips as a Library

This section describes using LIPS as a library. Most functions which are available in lisp are also available in C++.

8.1 Properties

This section describes some properties that are used internally by the shell.

`alias` this is used during alias expansion. Let's say we want to define an alias for ls that uses the -F option of ls: `(putprop 'l 'alias '(ls -F))`. To simplify defining aliases the lisp function alias is provided (see below).

`autoload` If during evaluation of a form, an undefined function is found the offending atom may have a file stored under this property. If this is the case the file is loaded and evaluation is allowed to continue. If the function is still undefined error processing takes over.