

Hedgehog Lisp 2.0.0

Table of Contents

1	Introduction	1
2	Getting Started	1
3	Lisp basics	2
4	Memory management and list processing	3
5	User defined functions	4
6	Exception handling	7
7	Integers	7
8	Strings	8
9	Conditional Compilation	8
10	Macros	9
11	Debugging	10
12	Profiling	10
13	Function reference	11
13.1	Special forms	11
13.2	Boolean arithmetic	12
13.3	List processing	12
13.4	AVL-trees	13
13.5	Predicates	15
13.6	Comparison functions	16
13.7	Arithmetic	16
13.8	String processing	18
13.9	Miscellaneous	20
13.10	UNIX	20
14	Library functions	26
14.1	Control structures	27
14.2	Unit testing	27
14.3	Operating system services	28
14.4	AVL-trees	38
14.5	A Python-like dictionary	38
14.6	Manipulating byte orders and integers in C structs as Lisp strings.	40
14.7	IP	41
14.8	List processing library functions	42
14.9	Math	43
14.10	Miscellaenous	44
14.11	Queues	44
14.12	String operations	45
14.13	State machines	46

1 Introduction

Hedgehog Lisp is the in-house development tool Oliotalo uses for embedded development. It is meant to be customizable to various embedded hardware platforms and to allow us to develop applications rapidly. The intention is that the Hedgehog Lisp interpreter, written in C, is ported to each target hardware platform, and applications are then written on top of this. This manual briefly teaches the language and an appendix provides a reference manual for the library. The reader is expected to know programming already.

An earlier generation of the Hedgehog platform consisted of a C library linked to application code. The library abstracted away most hardware dependencies. However, application programming in C is less productive and more error prone than in a high level language, as long as the high level language is capable of the task.

Additionally, Oliotalo has the need to update the applications over the air, which in some environments is difficult if the application is written in C. In these environments, not just the Hedgehog C library, but the whole operating system is linked to the application to form a single monolithic block. Updating the application would require loading whole block, which is more expensive to do and harder to implement.

Thus, Oliotalo decided to implement a simple, but powerful language and write its applications in this language. We chose to implement our own dialect of the Lisp language, since it is small and powerful as a language, and simple to implement.

Lisp is a one of the three oldest programming languages still in use. It was invented by John McCarthy in the late 1950s and has been in constant use since then. There have, through the years, been many versions of Lisp. At the moment the major versions are Common Lisp (standardized internationally), Scheme, and Elisp (used inside the Emacs editor). They are all similar, but different, and mostly share the same philosophy, even if they implement it differently. Hedgehog Lisp is its own dialect, and does not try to implement any existing dialect. This is so that we can concentrate on optimizing both the language and its implementation for our own purposes. However, all Lisp-like languages are similar enough that it is easy to pick up a new one; much easier than, say, picking up Java if you C, or vice versa.

Hedgehog Lisp is functional: no functions may have any side effects, except for the builtin I/O operations. The language was originally designed by Lars Wirzenius. The current implementation and some of the design is by Kenneth Oksanen.

2 Getting Started

It is easiest to install Hedgehog Lisp on a Debian GNU/Linux i386 system using the binary package (`hedgehog.deb`) provided by Oliotalo. On other platforms, you need to get the source package and compile it yourself.

The Hedgehog Lisp implementation consists of a compiler and interpreter. The compiler, `hhc`, generates bytecode, which the interpreter, `hhi`, then runs. To compile a Hedgehog Lisp program to byte code, you write the source code to a file and run the following command:

```
hhc foo.hl
```

where *foo.hl* is the name of your file. Try out for example *fib.hl* for the ubiquitous Fibonacci-test. You can find it in the source package or in `/usr/share/doc/hedgehog/examples` if you installed the binary package.

The compiler writes out two output files, *fib.hlo* for the byte code program and *fib.hls* for the byte code assembler. The latter one is useful only for debugging, and we shall return to that later.

To run the byte code program, run the command:

```
hhi
```

The Hedgehog Lisp interpreter is not suitable for using interactively: it reads in the whole input file before interpreting it, so there is no read-eval-output loop as in traditional Lisp interpreters.

Example: Simple expression.

```
(print (+ 1 2 3) "\n")  
  
hhc foo.hoglisp  
hhi  
6
```

Pass `hhc` and `hhi` the flag `--help` or `-h` to see their full command line interface. Note that the sizes of the heap and stack sizes are fixed during each execution, but are adjustable from the command line.

3 Lisp basics

A Hedgehog Lisp program consists of a sequence of expressions. Some expressions compute things, others define functions, which later expressions can call. An expression can be an *atom*, i.e., an atomic piece of data, such as an integer. An expression can also be a list, i.e., a sequence of expressions inside parentheses. The example above is one list expression, which contains four atomic expressions: the symbol `+` and the integers 1, 2, and 3.

Atoms can be symbols, integers, or strings. We will return to strings later in this tutorial, and concentrate on symbols and integers for now.

A symbol name may not contain whitespace or parentheses, but may otherwise contain anything. For example, a plus sign is quite a valid name; it is not being treated specially. A symbol name may be *bound* to a value, such as an atom, a list, or a function.

When a list expression is evaluated, it is treated as a function call. The first value in the list identifies the function to be called, the remaining values are its arguments. Typically, as in the example above, the first value in the list is a symbol that is bound to a function value. In the example, the `+` symbol is bound to a *built-in function* that computes the sum of its arguments (which must all be integers).

Before the function is called, its arguments are evaluated. In the example, each argument is an integer, and its value is itself, so there is not much to compute. In the general case, however, the argument might itself be a call to a function, as in the next example.

Example: Nested expression.

```
(+ 1 (+ 2 3))
```

This example contains two list expressions. The inner one `(+ 2 3)` is evaluated first, and returns 5. This is then passed as the second argument to the outer one; in effect, the outer call becomes `(+ 1 5)`.

4 Memory management and list processing

Data is stored by the Hedgehog Lisp interpreter in *cells* of different types. A cell can contain an atom: a symbol, an integer, a string, or a built-in function. Lists are constructed using a special cell called a *cons* containing references to two other cells. A simple linked list consists of a sequence of cons cells, where the first reference points at the value at the node in the list, and the second one points at the next node in the list.

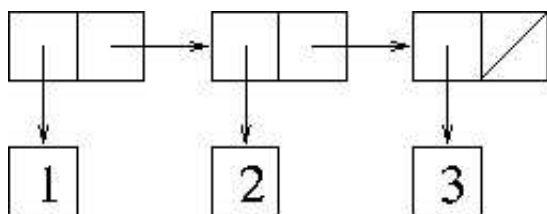


Figure: Simple linked list

A new cons cell is constructed with the built-in function `cons`:

```
(cons 1 (cons 2 (cons 3 nil)))
```

This constructs a list of the numbers 1, 2, and 3, such as the one in figure. `nil` is an atom bound to an empty reference, corresponding to null pointer in C.

Constructing lists with `cons` quickly becomes tedious. It would be easier to be able to write the list out directly:

```
(1 2 3)
```

However, this is evaluated as if it were a function call (and thus resulting in an error). To avoid this, use the special `quote` function, which prevents evaluation of its arguments:

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

This results in a value of `(1 2 3)`. Note that if `quote` were a normal function, its argument would be evaluated before it is called.

Because `quote` is used so often, it can be abbreviated with an apostrophe:

```
'(1 2 3)
```

Alternatively, if you want to construct a list from computed values rather than constants, you can use `list`:

```
(list (+ 1 2) 3)
```

`list` returns its arguments (the values of its argument expressions) in a list, in this case the return value would be `(3 3)`.

Once we have a list value (as opposed to a list expression), we can operate on it using the built-in `car` and `cdr` functions. A list is represented by its leading cons cell, and `car` and `cdr` return the first and second reference, respectively. Thus, to extract the second element of the list `(1 2 3)`, you would write the following:

```
(car (cdr '(1 2 3)))
```

It is instructive to see how this is evaluated. First, the list `(1 2 3)` is constructed and passed to `cdr`, which returns the second reference in the leading cons cell. This means that the value of the `cdr` call is a reference to the second cons cell in the list; in effect, `cdr` returns the list `(2 3)`. This is then passed to `car`, which returns the first reference in the leading cons cell, i.e., a reference to the integer 2. This is then the result of the whole expression.

The `cons` function allocates a cell explicitly. There is no explicit memory de-allocation. Instead, Hedgehog Lisp relies on garbage collection: cells that are no longer referred to by anyone are destroyed and re-used invisibly to the programmer.

5 User defined functions

Hedgehog Lisp provides two ways for the programmer to define a new function: `def` and `fn`.

Example: Simple function.

```
(def (addone x)
  (+ x 1))
```

Here, we define a new function, called `addone`, that takes one argument (`x`) and returns as its value the value of the argument plus one.

Example: Compute absolute value of a number.

```
(def (abs x)
  (cond
    (< x 0) (- 0 x)
    x))
```

In the second example, we also introduce the special function `cond`, which implements conditionals. There are two forms of `cond`:

- `(cond p1 e1 ... pn en edefault)`
- `(cond p1 e1 ... pn en)`

In the first form, `cond` gets an odd number of expressions, and evaluates each `pi` in order, until it finds one that is true. It then evaluates the corresponding `ei`, and returns that as its value. The remaining expressions are not evaluated. If no `pi` is true, `cond` evaluates `edefault` as its value. In the second form, where no default value is given, `cond` returns `nil`.

A value is considered to be false, if it is `nil`, or the integer zero, or the empty string. Otherwise it is considered to be true.

Example: Compute length of a list.

```
(def (list-length x)
  (cond
    (x (+ 1 (list-length (cdr x))))
    (0)))
```

This function returns the length of its argument, which must be a list. It uses recursion and `cdr` to traverse the list, and counts the number of elements. Unfortunately, it also uses function call stack space proportional to the length of the list. If the list is very long, it will result in much space being used. This is not necessary. A more space efficient way to write the function is to write it so that it only tail recursion, i.e., the result of the recursive call is directly the result of the function.

Example: Tail recursion.

```
(def (list-length-helper remaining n)
  (cond
    (remaining (list-length-helper (cdr remaining) (+ 1 n)))
    (n)))

(def (list-length x)
  (list-length-helper x 0))
```

The `list-length-helper` function only uses tail recursion. This is important, because Hedgehog Lisp carefully optimizes tail recursion calls (indeed, all tail calls) so that they don't use extra stack space. In other words, when you traverse a list with tail recursion, it doesn't take any extra space. Tail recursion is the equivalent of iteration in imperative programming languages. Since this is such an important concept, there is a special form, `tailcall`, which does not affect the computation in any way, but causes a compile-time error if it is used in a non-tailrecursive position in the program code.

```
(def (list-length-helper remaining n)
  (cond
    (remaining
     (tailcall (list-length-helper (cdr remaining) (+ 1 n))))
    (n)))
```

Example: Local functions.

```
(def (list-length x)

  (def (helper remaining n)
    (cond
      (remaining (helper (cdr remaining) (+ 1 n)))
      (n)))

  (helper x 0))
```

Here we have defined the helper function as a local function inside `list-length` itself. This makes it obvious to others reading the code that the helper function is used only by the `list-length` function, and is not meant to be used by others. It also reduces the amount of name space clutter by helper functions, of which there are going to be many.

Example: Functions as arguments.

```
(def (find-item items is-this-it)

  (cond
    (not items) nil
    (is-this-it (car items)) (car items)
    (find-item (cdr items) is-this-it)))
```

This function is used to find an item in a list. Since the algorithm for traversing a list and finding an item is not dependent on how the desired item is identified, `find-item` gets a function to make the decision as its second argument. This way, regardless of how we want to choose an item, we can always use the same `find-item` function.

Example: Searching in a list.

```
(def (is-arthur person)
  (= (car person) 12))

(find-item '((12 "Arthur Dent") (42 "Ford Prefect")) is-arthur)
```

Here we use `find-item` to find in a list of lists one where the first element is 12. Each sublist represents a person, with the first element of the list giving the person's unique code number.

It would be useful to have a generic function for finding a particular person, of course. To do this, we must construct a function at run time that matches our desired person. This can be done with local functions and lexical scoping rules.

Example: Finding anyone in a list.

```
(def (find-person persons id)
  (def (is-person person)
    (= (car person) id))
  (find-item persons is-person))
```

The local function `is-person` inside `find-person` is defined anew for each call of `find-person`. Hedgehog Lisp uses lexical scoping, which means that `is-person` can refer to the value of `find-person`'s arguments (and other local function names), in addition to globally defined function names. That is, when `is-person` is called, and refers to the value of `id`, the value it gets is the value of `id` for the `find-person` call that defined that particular instance of `is-person`. Therefore, `is-person` returns true for exactly the desired person.

It is not necessary to use a local `def` to define a function such as `is-helper`. Often it is not necessary for the helper function to have a name, and it would be more convenient not to have to first define the function and then to refer to it by name. Lisp languages thus can define nameless functions with an operation typically called lambda, which Hedgehog Lisp calls `fn` (for no good reason, except that it is shorter; this idea was picked up from Paul Graham's description of the Arc language).

Example: Lambda function.

```
(def (find-person persons id)
  (find-item persons
    (fn (person)
      (= (car person) id))))
```

This example is arguably clearer than the previous one, and does the same thing.

Example: Variable argument functions.

```
(def (list ... args)
  args)

(list 1 2 3 4 5 6)
```

The function `list` gets some number of expressions as its argument. It returns a list with the values of those expressions. When a function is defined so that its last argument is preceded with ellipsis `...`, it gets its arguments packaged in a list. If it has more than one formal argument, those are assigned values from the actual arguments at call time, and the remaining ones are put into a list and that is assigned to the last argument.

6 Exception handling

The builtin function `throw` throws an exception. Exceptions are represented by symbols (not the values the symbols are bound to). The function `catch` catches an exception. If an exception is not caught, the interpreter terminates the program.

Example: Exception handling

```
(def (divide a b)
  (cond
    (= b 0)
    (throw divide-by-zero)
    (/ a b)))

(catch
  (divide 10 0)
  divide-by-zero 0)
```

In this example we throw an exception if the caller tries to divide by zero, and then in the caller we catch the exception.

When memory is about to exhaust, the interpreter automatically throws a `out-of-memory-exception`. If it is caught, it collects garbage again and tries to continue. Otherwise the interpreter exits with a special error code.

7 Integers

Integers are 32 bits large and signed. All integer constants are decimal, unless prefixed by `0x`, in which case they are hexadecimal. Basic mathematical operations (`+`, `-`, `*`, `/`) are available, as are remainder (`%`) and power (`pow`). Comparisons (`<`, `<=`, `=`, `!=`, `>`, `>=`) are also available, of course.

Bitwise integer operations (`|`, `&`, `^`, `<<` and `>>`), however, treat their argument as an unsigned value.

8 Strings

Strings are immutable: once created, they can't be changed. They are thus similar in this regard to all other data structures in Hedgehog Lisp, none of which are mutable.

String literals use a C-like syntax: they are surrounded by double quotes and backslash is used as an escape character. The escape sequences are listed in the following table.

Table: Escape sequences in string literals.

<code>\\</code>	backslash (<code>\</code>)
<code>\"</code>	double quote (<code>"</code>)
<code>\a</code>	audible bell
<code>\b</code>	backspace
<code>\f</code>	form feed
<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\t</code>	tab
<code>\v</code>	vertical tab
<code>\xnn</code>	character whose hexadecimal code is <i>nn</i>
<code>\nnn</code>	character whose octal code is <i>nnn</i>

Strings can be manipulated with functions listed in the library reference.

9 Conditional Compilation

Conditional compilation makes it possible to have alternate implementations for some functionality and choose the implementation for example from the compilers's command line with the flag `-D`.

The facility is very similar to the C preprocessor's `#ifdef`, `#ifndef`, `#else`, and `#endif`. Note, however, that macros have no value except whether they are defined or not. Therefore there is no `#if`, and `#define` is followed by the macro name, but nothing more.

Example: Conditional Compilation.

```

#define QUIET
...
#ifndef QUIET
  (def (print-to-console s)
    (print s))
#else
  (def (print-to-console s)
    nil)
#endif
...

```

`#ifdef`s can be nested. Some symbols, such as `HH_UNIX` may be predefined by the compiler.

10 Macros

Since user-defined functions always evaluate their arguments before issuing the call, they have limits in their capabilities of extending the syntax of the language. Macros, on the other hand, are always expanded during compilation time, without evaluating their arguments.

Note: Like in many other languages, macros are a brittle and possibly difficult concept, and best be avoided by the novice programmer.

Macros are essentially rewrite rules defined by the special form `def-syntax`. The first argument is a *pattern*, which is matched against all expressions in the program code after the definition of the macro. The pattern may be an arbitrary syntactic construct containing for atoms, lists, pattern variables and ellipsis (`. . .`). Pattern variables are special symbols that begin with a question mark. They match any expression, but a pattern variable may appear only once in the pattern. Ellipsis followed by a pattern variable means that the rest of the list is stored in the pattern variable, just like in variable argument function definitions.

The second argument of `def-syntax` is the *replacement*. Any pattern variable that was bound during the matching, will be substituted with that value. Unbound pattern variables are bound to globally unique symbols, thereby providing a mechanism for generating temporary variables.

Example: Simple macros.

```

(def-syntax (assert ?x)
  (if (not ?x)
      (panic "Failed assertion `" (quote ?x) "'.\nExiting.\n")))

```

Note that the pattern variable is instantiated also inside of `quote`. This would not be the case if `assert` were a normal function.

If the macro body contains expressions starting with `##`, the remaining arguments are concatenated together into a new symbol. For example, `(## make- ?thing)`, where `?thing` is bound to `love`, would be rewritten to the symbol `make-love`.

Another special symbol, `#'`, is used to prevent the macro expansion of its argument. This is useful when the macro body expands into new macro definitions to prevent a premature expansion of the new macro variables.

If the macro is expanded on the top-level, in a function body or do-expression, then the rewrite may contain several expressions, all of which are expanded to the same level as the macro was used.

The macro facility is still improving. In addition to bugfixes and improved error messages, one requested feature is the extraction of source code location of macro arguments, etc.

11 Debugging

Since `hhi` is intended to be as small as possible, it does not contain a featureful debugger. It does print information on why and in which instruction the byte code program failed.

Example: Debugging.

```
$ cat bar.lisp
(+ 1 "foo")
$ hhc bar.lisp
$ hhi
Fatal program error #4: Expected an integer. pc = 000006, sp = 0.
$ grep 000006 hh.asm
000006 ("bar.lisp":1)          add
```

In other words, the program failed because of a type error in addition on line 1 in file `bar.lisp`.

If the symbol `HH_SMALL` is not defined during compilation, an approximate function call sequence (backtrace) is printed out when the program fails. Here too the return addresses are shown as zero-padded, six digits wide program counter values, which have to be looked up from the corresponding `.hls` file. Tail-recursive calls are naturally not shown in the function call list.

12 Profiling

If the symbol `HH_TESTING` is defined during compilation, it is possible to use a rudimentary profiling mechanism: `hhi` can be given the flag `-p`, and after the program exits, it prints out a list of all program counters and the number of times they were executed.

Example: Profiling.

```
$ hhc tests/bench-sm.hl
$ hhi -p |& grep '^0' | sort -k 2 -n -r -s | head
001827    13468
001828    13468
001830    13468
001831    13468
...
$ grep 001827 bench-sm.hls
001827 ("prelude.lisp":263)          push
```

In other words, the most frequently executed instructions were in the function `nth` in the prelude.

13 Function reference

The functions in this section are defined inside the compiler.

13.1 Special forms

`(set variable value)`

Create and bind a local variable. May appear only in an expression sequence, such as ‘do’ or in the body of ‘def’, ‘fn’, or the top-level. The variable is valid only during the rest of the sequence, not before the ‘set’, or outside the surrounding expression sequence.

- *variable* (symbol): Variable name.
- *value* (any): Bound value.

`(quote form)`

Quote, i.e. prevent the given form from being evaluated.

- *form* (any): Form.

`(if cond then else)`

Evaluate either branch depending on the condition.

- *cond* (any): Condition.
- *then* (any): Evaluated if condition true.
- *else* (any): Evaluated if condition false (optional).

`(do bodies)`

Execute all expressions in a sequence, return last value.

- *bodies* (any, optional): Expressions to evaluate.

`(catch body tag catcher)`

Catch an exception.

- *body* (any): Protected body.
- *tag* (symbol): Catch tag.
- *catcher* (any): Exception handler.

`(throw tag)`

Throw an exception.

- *tag* (symbol): Catch tag.

(*apply function arguments*)

Call the given function with the given argument list. Note that even if the apply would be in a tail-recursive position, the call to the given function is not a tail-call.

- *function* (any): Function to call.
- *arguments* (cons): Argument list.

(*tailcall expr*)

Cause compile-time error if this is not a tail-recursive position.

- *expr* (any): Any expression.

13.2 Boolean arithmetic

(*and values*)

Boolean and.

- *values* (any, optional): Values interpreted as booleans.

(*or values*)

Boolean or.

- *values* (any, optional): Values interpreted as booleans.

(*not value*)

Boolean not.

- *value* (any): Value interpreted as boolean.

13.3 List processing

(*cons left right*)

Create a new cons cell.

- *left* (any): Value for left slot in cons cell.
- *right* (any): Value for right slot in cons cell.

(*car cell*)

Return left slot of the cons cell argument.

- *cell* (cons): Cons cell.

(cdr *cell*)

Return right slot of the cons cell argument.

- *cell* (cons): Cons cell.

(tuple-arity *x*)

Return the number of fields in the tuple, or 0 if not a tuple.

- *x* (any): Value

(tuple-make *values*)

Create a tuple of the given values.

- *values* (any, optional): Values.

(tuple-make-from-list *list*)

Create a tuple of the given list of values.

- *list* (cons): List of tuple field values.

(tuple-index *tuple i*)

Read the contents of the given field in the given tuple.

- *tuple* (any): A sufficiently wide tuple
- *i* (integer): Field number, starting from zero

(tuple-with *tuple i value*)

Copy the tuple with a new value in the specified slot.

- *tuple* (any): A sufficiently wide tuple
- *i* (integer): Field number, starting from zero
- *value* (any): The new value

13.4 AVL-trees

(avl-make-node *key value left right*)

Make a new AVL-tree node with the given key, value and subtrees. If the height difference of the subtrees is two, the routine performs necessary rotations to bring the new node into balance. No rotations are made if the heights differ less, and a fatal error is raised if the heights differ by three or more.

- *key* (any): Key
- *value* (any): Value
- *left* (AVL-tree node): Left subtree
- *right* (AVL-tree node): Right subtree

(*avl-height node*)

Get the height of the given AVL-tree node, zero for nil or for something not an AVL-tree node.

- *node* (AVL-tree node): AVL-tree node

(*avl-key node*)

Get the key in the given AVL-tree node.

- *node* (AVL-tree node): AVL-tree node

(*avl-value node*)

Get the value in the given AVL-tree node.

- *node* (AVL-tree node): AVL-tree node

(*avl-left node*)

Get the left subtree of the given AVL-tree node.

- *node* (AVL-tree node): AVL-tree node

(*avl-right node*)

Get the right subtree of the given AVL-tree node.

- *node* (AVL-tree node): AVL-tree node

(*default-cmpfun a b*)

A simple default comparison function for symbols, integers, and strings. Returns -1, 0, or 1 if 'a' is considered to be less than, equal to, or greater than 'b', respectively. Integers are considered less than all symbols, which in turn are considered less than strings.

- *a* (any): First comparand
- *b* (any): Second comparand

(*default-avl-get tree key default_value*)

A more efficient implementation of AVL tree searching assuming the default key comparison function.

- *tree* (AVL-tree node): AVL-tree
- *key* (any): Key to be searched
- *default_value* (any): Value returned when key not found

(default-avl-put *tree key value*)

A more efficient implementation of AVL tree insertion/replacement assuming the default key comparison function.

- *tree* (AVL-tree node): AVL-tree
- *key* (any): Key to be searched
- *value* (any): Value to be inserted/replaced

13.5 Predicates

(int? *value*)

Is argument an integer?

- *value* (any): Value of any type.

(string? *value*)

Is argument a string?

- *value* (any): Value of any type.

(symbol? *value*)

Is argument a symbol?

- *value* (any): Value of any type.

(fn? *value*)

Is argument a function?

- *value* (any): Value of any type.

(cons? *value*)

Is argument a cons cell?

- *value* (any): Value of any type.

(eq? *a b*)

Are two values, of any type, equal?

- a (any): First value.
- b (any): Second value.

13.6 Comparison functions

($<$ $ints$)

Is each argument less than the following one?

- $ints$ (integer, optional): Integers.

($<=$ $ints$)

Is each argument less than or equal to the following one?

- $ints$ (integer, optional): Integers.

($=$ $ints$)

Is each argument equal to the following one?

- $ints$ (integer, optional): Integers.

($>=$ $ints$)

Is each argument greater than or equal to the following one?

- $ints$ (integer, optional): Integers.

($>$ $ints$)

Is each argument greater than the following one?

- $ints$ (integer, optional): Integers.

($!= a b$)

Do the two arguments have different values?

- a (integer): First value.
- b (integer): Second value.

13.7 Arithmetic

($+$ $ints$)

Add arguments together.

- $ints$ (integer, optional): Integer arguments.

(- *ints*)

Subtract other arguments from first one, or negate if only one.

- *ints* (integer, optional): Integer arguments

(* *ints*)

Multiply arguments together.

- *ints* (integer, optional): Integer arguments.

(/ *ints*)

Divide arguments with each other.

- *ints* (integer, optional): Integer arguments.

(% *a b*)

Return remainder (a modulo b).

- *a* (integer): The numerator.
- *b* (integer): The denominator.

(& *ints*)

Bitwise and arguments together.

- *ints* (integer, optional): Integer arguments.

(| *ints*)

Bitwise or arguments together.

- *ints* (integer, optional): Integer arguments.

(~ *value*)

Bitwise not.

- *value* (integer): Value to be inverted.

(^ *ints*)

Bitwise XOR.

- *ints* (integer, optional): Integer arguments.

(<< *value n*)

Bitwise left shift.

- *value* (integer): Value to be shifted.
- *n* (integer): Number of bits to shift.

(*>> value n*)

Bitwise right shift.

- *value* (integer): Value to be shifted.
- *n* (integer): Number of bits to shift.

13.8 String processing

(*strlen str*)

Return length of string argument.

- *str* (string): The string.

(*substr str pos len*)

Copy part of existing string into new string.

- *str* (string): Original string.
- *pos* (integer): Substring position in STR.
- *len* (integer): Substring length, -1 means to end of STR.

(*strcmp str1 str2*)

Compare two strings: return negative if first is less than second, positive if greater, zero if equal.

- *str1* (string): First string.
- *str2* (string): Second string.

(*ord ch*)

Return character code of first character in argument.

- *ch* (string): The character (non-empty string).

(*chr code*)

Return string containing character whose code is argument.

- *code* (integer): The character code.

(*strcat strings*)

Catenate strings.

- *strings* (string, optional): Strings to be catenated.

(*atoi str base*)

Convert a string to an integer.

- *str* (string): The string to convert.
- *base* (integer): Base for the number in STR.

(*itoa value base*)

Convert an integer to a string.

- *value* (integer): Integer to convert to string.
- *base* (integer): Base for the conversion.

(*symboltostring symbol*)

Convert the name of a symbol to its string representation.

- *symbol* (symbol): The symbol to convert.

(*strstr string pattern*)

Return position of pattern inside string, or -1 for not found.

- *string* (string): String to search inside.
- *pattern* (string): String to search for.

(*strrstr string pattern*)

Return last position of pattern inside string, or -1 for not found.

- *string* (string): String to search inside.
- *pattern* (string): String to search for.

(*strsplit-last string sep*)

If SEP occurs in STRING, return a cons where car and cdr are strings split at the last occurrence of SEP (as if searched by strrstr). Otherwise return nil.

- *string* (string): String to be split.
- *sep* (string): Separator.

(*hex string*)

Encode bytes in a string with hexadecimals.

- *string* (string): String to convert.

13.9 Miscellaneous

(*print values*)

Write arguments to the default destination (stdout, serial, log).

- *values* (any, optional): Values to print.

(*snprint max_length accuracy value*)

Write last argument to a string of specified length and given accuracy.

- *max_length* (integer): Maximum string length, use -1 for unlimited.
- *accuracy* (integer): Depth or accuracy of the print.
- *value* (any): Value to print.

(*panic strings*)

Write panic message to log and terminate program.

- *strings* (string, optional): Values to output.

(*available-mem*)

Return the number of free bytes (not including garbage).

(*gc*)

Do a hard garbage collection. Collect all garbage. This may take a some time, so you should call this only when you know the program has nothing interesting to do. The Hedgehog Lisp implementation collects garbage as necessary, so it is never necessary to call this function. If you want to clear out large amounts of garbage at once, and have nothing better to do, then do call this function.

(*hedgehog-version*)

Return a string giving the version of the Hedgehog LISP implementation. The version consists of three decimal integers separated by dots. In CVS versions, the integers may be negative.

13.10 UNIX

(*unix-gettimeofday*)

Return the current time, as a cons of seconds and microseconds since the beginning of an unspecified epoch. See `gettimeofday(2)`.

`(unix-fork)`

Perform UNIX fork.

`(unix-exec filename argv envp)`

Perform UNIX `execve`. The first and second argument have identical meaning to `execve` (except we expect list of strings). The third argument must currently be an empty list, but may in future designate changes in the environment.

- *filename* (string): Program to execute
- *argv* (cons): List of strings passed to the program as argv
- *envp* (cons): Environment, but nil for now

`(unix-dup2 from_fd to_fd)`

Call the Unix `dup2` system call. Return -1 for error.

- *from_fd* (integer): Original file descriptor.
- *to_fd* (integer): New file descriptor.

`(unix-close fd)`

Close an open file descriptor on Unix.

- *fd* (integer): The file descriptor.

`(unix-select secs usecs)`

Perform UNIX `select`.

- *secs* (integer): Max wait time seconds,
- *usecs* (integer): and milliseconds.

`(unix-add-to-read-fds fd)`

Add the given *fd* to the next `select(2)`'s read fd set.

- *fd* (integer): File descriptor.

`(unix-add-to-write-fds fd)`

Add the given *fd* to the next `select(2)`'s write fd set.

- *fd* (integer): File descriptor.

(unix-fd-is-readable *fd*)

Is the given *fd* readable according to the latest select(2).

- *fd* (integer): File descriptor.

(unix-fd-is-writable *fd*)

Is the given *fd* writable according to the latest select(2).

- *fd* (integer): File descriptor.

(unix-clr-fdsets)

Clear the *fd* sets in preparation for the next select(2).

(unix-dir-list *dirname*)

Return a list of file names in the given directory.

- *dirname* (string): Directory name.

(unix-unlink *filename*)

Delete the given file, return true on success.

- *filename* (string): File name.

(unix-open *filename flags*)

See open(2).

- *filename* (string): Name of file to open.
- *flags* (integer): Bitwise-or'ed mode flags.

(unix-socket *domain type protocol*)

See socket(2).

- *domain* (integer): Communication domain.
- *type* (integer): Communication type.
- *protocol* (integer): Communication protocol.

(unix-setsockopt *socket level optname optval*)

See setsockopt(2).

- *socket* (integer): File descriptor.

- *level* (integer): level, see `setsockopt(2)`.
- *optname* (integer): *optname*, see `setsockopt(2)`.
- *optval* (string): *optval*, see `setsockopt(2)`.

(`unix-fcntl` *fd cmd data*)

See `fcntl(2)`.

- *fd* (integer): File descriptor.
- *cmd* (integer): Command, see `setsockopt(2)`.
- *data* (any): nil or an integer or string argument.

(`unix-connect` *fd sockaddr*)

See `connect(2)`.

- *fd* (integer): Socket.
- *sockaddr* (string): Address to connect.

(`unix-bind` *fd sockaddr*)

See `bind(2)`.

- *fd* (integer): Socket.
- *sockaddr* (string): Address to bind to.

(`unix-listen` *fd backlog*)

See `listen(2)`.

- *fd* (integer): Socket.
- *backlog* (integer): Length of queue for pending connections.

(`unix-accept` *fd*)

See `accept(2)`. Returns either nil in case of failure, or a cons of the accepted file descriptor socket and possibly its *sockaddr*.

- *fd* (integer): Listening socket.

(`unix-read` *fd nbytes*)

Read the given number of bytes from the given file descriptor.

- *fd* (integer): File descriptor.

- *nbytes* (integer): Number of bytes to read.

(`unix-write` *fd data*)

Write the string to the given file descriptor.

- *fd* (integer): File descriptor.
- *data* (string): String to write.

(`unix-recvfrom` *socket nbytes flags*)

Read at most the given number of bytes from the given socket. On failure returns nil, on success a tuple containing the read data and its source address. See `recvfrom(2)`.

- *socket* (integer): File descriptor.
- *nbytes* (integer): Number of bytes to read.
- *flags* (integer): Flags, see `recvfrom(2)`.

(`unix-sendto` *socket data flags sockaddr*)

Write the given bytes to the given socket. See `sendto(2)`.

- *socket* (integer): File descriptor.
- *data* (string): Data to send.
- *flags* (integer): Flags, see `sendto(2)`.
- *sockaddr* (string): Sockaddr, see `sendto(2)`.

(`unix-lseek` *fd offset whence*)

Reposition read/write file offset. See `lseek(2)`.

- *fd* (integer): File descriptor.
- *offset* (integer): Offset.
- *whence* (integer): `SEEK_SET`, `SEEK_CUR` or `SEEK_END`.

(`unix-mount` *source target fstype flags data*)

See `mount(2)`.

- *source* (string): Source (e.g. device).
- *target* (string): Target (mount point).
- *fstype* (string): File system type, e.g. `vfat`.

- *flags* (integer): Flags.
- *data* (string): Addition fstype-dependent data, or nil.

(unix-umount *target*)

See umount(2).

- *target* (string): Target (mount point).

(unix-usleep *usecs*)

Sleep for the specified number of microseconds. Note that this issues UNIX usleep, which suspends the whole lisp program for the corresponding duration. Therefore, use only for very short sleeps!

- *usecs* (integer): Number of microseconds to sleep.

(unix-errno)

Return the value of UNIX errno.

(unix-tcgetattr *fd*)

See tcgetattr(3). Returns the termios struct as a Lisp string.

- *fd* (integer): File descriptor.

(unix-tcsetattr *fd optional_actions termios*)

See tcsetattr(3).

- *fd* (integer): File descriptor.
- *optional_actions* (integer): See tcsetattr(3).
- *termios* (string): Termios struct as a Lisp string.

(unix-tcsendbreak *fd duration*)

See tcsetattr(3).

- *fd* (integer): File descriptor.
- *duration* (integer): See tcsetattr(3).

(unix-tcdrain *fd*)

See tcsetattr(3).

- *fd* (integer): File descriptor.

(unix-tcflush *fd queue_selector*)

See `tcflush(3)`.

- *fd* (integer): File descriptor.
- *queue_selector* (integer): See `tcflush(3)`.

(`unix-tcflow` *fd action*)

See `tcflow(3)`.

- *fd* (integer): File descriptor.
- *action* (integer): See `tcaction(3)`.

(`unix-cfmakeraw` *termios*)

See `cfmakeraw(3)`. Returns the raw's termios struct as a Lisp string.

- *termios* (string): Unraw'd termios struct as a Lisp string.

(`unix-cfgetispeed` *termios*)

See `cfgetispeed(3)`.

- *termios* (string): Termios struct as a Lisp string.

(`unix-cfgetospeed` *termios*)

See `cfgetospeed(3)`.

- *termios* (string): Termios struct as a Lisp string.

(`unix-cfsetispeed` *termios speed*)

See `cfsetispeed(3)`. Returns a new termios struct with the given speed.

- *termios* (string): Termios struct as a Lisp string.
- *speed* (integer): See `cfsetispeed(3)`.

(`unix-cfsetospeed` *termios speed*)

See `cfsetospeed(3)`. Returns a new termios struct with the given speed.

- *termios* (string): Termios struct as a Lisp string.
- *speed* (integer): See `cfsetospeed(3)`.

14 Library functions

The functions in this section are defined in the prelude. From the application programmer's view, there is no difference, but they are separate from the builtin functions in the manual due to stupid production reasons.

14.1 Control structures

The following control structures are implemented at the Lisp level, using macros.

`(cond ?c ?t ... ?r)`

`(cond predicate_1 expr_1 predicate_2 expr_2 ... default_expr)` is a replacement for multiple 'if' expressions within each other.

`(let ?name ?value ... ?rest)`

`(let symbol_1 expr_1 symbol_2 expr_2 ... expr)` is a replacement for multiple `(set foo bar)` sequences. `expr_i` are evaluated and bound to `symbol_i` in sequence, and the bindings are not visible outside the `let`. Note that there are fewer parentheses than in most Lisp dialects.

Example: `(let foo 1 bar 2 (pr (+ foo bar)))` prints out 3.

`(def-record ?name ... ?fields)`

`(def-record record_name field_1 value_1 ...)` defines a set of macros for accessing and updating fields mnemonically in a tuple. For example `(def-record abc a 1 b 2 c 3)` creates a record called `abc` containing three fields `a`, `b`, and `c` with default values of 1, 2, and 3, respectively. The record definition expands into three macros `abc-a`, `abc-b`, and `abc-c` for reading the fields of the record, and `abc-with-a`, `abc-with-b` etc. for copying the given record with the respective field bound to a new value. Furthermore a new value `abc-default` containing the default bindings of `abc` is generated.

`(equal? a b)`

Are two values equal?

The builtin `eq?` doesn't deal with lists, for example. This function returns true if `eq?` returns true for its arguments, or if both arguments are cons and `equal?` is true for both element pairs, or if the two arguments are structurally identical AVL-trees with `equal?` keys and values.

14.2 Unit testing

The functions in this section provide for a simple form of automatic unit testing. They are used extensively within the standard library implementation. Eventually it will be possible to turn running of the unit tests on and off with a compiler command line option. Always write your unit tests so that it doesn't matter for the execution of the library or application whether they are actually executed.

`(fail-unless ?expr)`

Abort program unless 'expr' is true.

(fail-unless-equal ?a ?b)

Abort program unless 'a' equals 'b'.

14.3 Operating system services

This section contains definitions automatically generated from the operating system's C include files.

unix-EPERM
unix-ENOENT
unix-ESRCH
unix-EINTR
unix-EIO
unix-ENXIO
unix-E2BIG
unix-ENOEXEC
unix-EBADF
unix-ECHILD
unix-EAGAIN
unix-ENOMEM
unix-EACCESS
unix-EFAULT
unix-ENOTBLK
unix-EBUSY
unix-EEXIST
unix-EXDEV
unix-ENODEV
unix-ENOTDIR
unix-EISDIR
unix-EINVAL
unix-ENFILE
unix-EMFILE
unix-ENOTTY
unix-ETXTBSY
unix-EFBIG
unix-ENOSPC
unix-ESPIPE
unix-EROFS
unix-EMLINK
unix-EPIPE
unix-EDOM
unix-ERANGE
unix-EDEADLK
unix-ENAMETOOLONG
unix-ENOLCK
unix-ENOSYS
unix-ENOTEMPTY
unix-ELOOP

unix-EWOULDBLOCK
unix-ENOMSG
unix-EIDRM
unix-ECHRNG
unix-EL2NSYNC
unix-EL3HLT
unix-EL3RST
unix-ELNRNG
unix-EUNATCH
unix-ENOCSI
unix-EL2HLT
unix-EBADE
unix-EBADR
unix-EXFULL
unix-ENOANO
unix-EBADRQC
unix-EBADSLT
unix-EDEADLOCK
unix-EBFONT
unix-ENOSTR
unix-ENODATA
unix-ETIME
unix-ENOSR
unix-ENONET
unix-ENOPKG
unix-EREMOTE
unix-ENOLINK
unix-EADV
unix-ESRMNT
unix-ECOMM
unix-EPROTO
unix-EMULTIHOP
unix-EDOTDOT
unix-EBADMSG
unix-EOVERFLOW
unix-ENOTUNIQ
unix-EBADFD
unix-EREMCHG
unix-ELIBACC
unix-ELIBBAD
unix-ELIBSCN
unix-ELIBMAX
unix-ELIBEXEC
unix-EILSEQ
unix-ERESTART
unix-ESTRPIPE
unix-EUSERS
unix-ENOTSOCK

unix-EDESTADDRREQ
unix-EMSGSIZE
unix-EPROTOTYPE
unix-ENOPROTOOPT
unix-EPRONOSUPPORT
unix-ESOCKTNOSUPPORT
unix-EOPNOTSUPP
unix-EPFNOSUPPORT
unix-EAFNOSUPPORT
unix-EADDRINUSE
unix-EADDRNOTAVAIL
unix-ENETDOWN
unix-ENETUNREACH
unix-ENETRESET
unix-ECONNABORTED
unix-ECONNRESET
unix-ENOBUFS
unix-EISCONN
unix-ENOTCONN
unix-ESHUTDOWN
unix-ETOOMANYREFS
unix-ETIMEDOUT
unix-ECONNREFUSED
unix-EHOSTDOWN
unix-EHOSTUNREACH
unix-EALREADY
unix-EINPROGRESS
unix-ESTALE
unix-EUCLEAN
unix-ENOTNAM
unix-ENAVAIL
unix-EISNAM
unix-EREMOTEIO
unix-EDQUOT
unix-ENOMEDIUM
unix-EMEDIUMTYPE
unix-O_ACCMODE
unix-O_RDONLY
unix-O_WRONLY
unix-O_RDWR
unix-O_CREAT
unix-O_EXCL
unix-O_NOCTTY
unix-O_TRUNC
unix-O_APPEND
unix-O_NONBLOCK
unix-O_NDELAY
unix-O_SYNC

unix-FASYNC
unix-F_DUPFD
unix-F_GETFD
unix-F_SETFD
unix-F_GETFL
unix-F_SETFL
unix-F_GETLK
unix-F_SETLK
unix-F_SETLKW
unix-F_SETOWN
unix-F_GETOWN
unix-FD_CLOEXEC
unix-F_RDLCK
unix-F_WRLCK
unix-F_UNLCK
unix-F_EXLCK
unix-F_SHLCK
unix-LOCK_SH
unix-LOCK_EX
unix-LOCK_NB
unix-LOCK_UN
unix-S_ISUID
unix-S_ISGID
unix-S_IRUSR
unix-S_IWUSR
unix-S_IXUSR
unix-S_IRWXU
unix-S_IREAD
unix-S_IWRITE
unix-S_IEXEC
unix-S_IRGRP
unix-S_IWGRP
unix-S_IXGRP
unix-S_IRWXG
unix-S_IROTH
unix-S_IWOTH
unix-S_IXOTH
unix-S_IRWXO
unix-ACCESSPERMS
unix-ALLPERMS
unix-DEFFILEMODE
unix-S_BLKSIZE
unix-S_IFMT
unix-S_IFDIR
unix-S_IFCHR
unix-S_IFBLK
unix-S_IFREG
unix-S_IFIFO

unix-S_IFLNK
unix-S_IFSOCK
unix-SHUT_RD
unix-SHUT_WR
unix-SHUT_RDWR
unix-SOCK_STREAM
unix-SOCK_DGRAM
unix-SOCK_RAW
unix-SOCK_RDM
unix-SOCK_SEQPACKET
unix-SOCK_PACKET
unix-PF_UNSPEC
unix-PF_LOCAL
unix-PF_UNIX
unix-PF_FILE
unix-PF_INET
unix-PF_AX25
unix-PF_IPX
unix-PF_APPLETALK
unix-PF_NETROM
unix-PF_BRIDGE
unix-PF_ATMPVC
unix-PF_X25
unix-PF_INET6
unix-PF_ROSE
unix-PF_DECnet
unix-PF_NETBEUI
unix-PF_SECURITY
unix-PF_KEY
unix-PF_NETLINK
unix-PF_ROUTE
unix-PF_PACKET
unix-PF_ASH
unix-PF_ECONET
unix-PF_ATMSVC
unix-PF_SNA
unix-PF_IRDA
unix-PF_PPPOX
unix-PF_WANPIPE
unix-PF_BLUETOOTH
unix-PF_MAX
unix-AF_UNSPEC
unix-AF_LOCAL
unix-AF_UNIX
unix-AF_FILE
unix-AF_INET
unix-AF_AX25
unix-AF_IPX

unix-AF_APPLETALK
unix-AF_NETROM
unix-AF_BRIDGE
unix-AF_ATMPVC
unix-AF_X25
unix-AF_INET6
unix-AF_ROSE
unix-AF_DECnet
unix-AF_NETBEUI
unix-AF_SECURITY
unix-AF_KEY
unix-AF_NETLINK
unix-AF_ROUTE
unix-AF_PACKET
unix-AF_ASH
unix-AF_ECONET
unix-AF_ATMSVC
unix-AF_SNA
unix-AF_IRDA
unix-AF_PPPOX
unix-AF_WANPIPE
unix-AF_BLUETOOTH
unix-AF_MAX
unix-SOL_RAW
unix-SOL_DECNET
unix-SOL_X25
unix-SOL_PACKET
unix-SOL_ATM
unix-SOL_AAL
unix-SOL_IRDA
unix-SOMAXCONN
unix-MSG_OOB
unix-MSG_PEEK
unix-MSG_DONTROUTE
unix-MSG_CTRUNC
unix-MSG_PROXY
unix-MSG_TRUNC
unix-MSG_DONTWAIT
unix-MSG_EOR
unix-MSG_WAITALL
unix-MSG_FIN
unix-MSG_SYN
unix-MSG_CONFIRM
unix-MSG_RST
unix-MSG_ERRQUEUE
unix-MSG_NOSIGNAL
unix-MSG_MORE
unix-INADDR_LOOPBACK

unix-INADDR_ANY
unix-INADDR_BROADCAST
unix-INADDR_NONE
unix-IPPROTO_IP
unix-IPPROTO_HOPOPTS
unix-IPPROTO_ICMP
unix-IPPROTO_IGMP
unix-IPPROTO_IPIP
unix-IPPROTO_TCP
unix-IPPROTO_EGP
unix-IPPROTO_PUP
unix-IPPROTO_UDP
unix-IPPROTO_IDP
unix-IPPROTO_TP
unix-IPPROTO_IPV6
unix-IPPROTO_ROUTING
unix-IPPROTO_FRAGMENT
unix-IPPROTO_RSVP
unix-IPPROTO_GRE
unix-IPPROTO_ESP
unix-IPPROTO_AH
unix-IPPROTO_ICMPV6
unix-IPPROTO_NONE
unix-IPPROTO_DSTOPTS
unix-IPPROTO_MTP
unix-IPPROTO_ENCAP
unix-IPPROTO_PIM
unix-IPPROTO_COMP
unix-IPPROTO_SCTP
unix-IPPROTO_RAW
unix-VINTR
unix-VQUIT
unix-VERASE
unix-VKILL
unix-VEOF
unix-VTIME
unix-VMIN
unix-VSWTC
unix-VSTART
unix-VSTOP
unix-VSUSP
unix-VEOL
unix-VREPRINT
unix-VDISCARD
unix-VWERASE
unix-VLNEXT
unix-VEOL2
unix-IGNBRK

unix-BRKINT
unix-IGNPAR
unix-PARMRK
unix-INPCK
unix-ISTRIP
unix-INLCR
unix-IGNCR
unix-ICRNL
unix-IUCLC
unix-IXON
unix-IXANY
unix-IXOFF
unix-IMAXBEL
unix-OPOST
unix-OLCUC
unix-ONLCR
unix-OCRNL
unix-ONOCR
unix-ONLRET
unix-OFILL
unix-OFDEL
unix-NLDLY
unix-NL0
unix-NL1
unix-CRDLY
unix-CR0
unix-CR1
unix-CR2
unix-CR3
unix-TABDLY
unix-TAB0
unix-TAB1
unix-TAB2
unix-TAB3
unix-BSDLY
unix-BS0
unix-BS1
unix-FFDLY
unix-FF0
unix-FF1
unix-VTDLY
unix-VT0
unix-VT1
unix-XTABS
unix-CBAUD
unix-B0
unix-B50
unix-B75

unix-B110
unix-B134
unix-B150
unix-B200
unix-B300
unix-B600
unix-B1200
unix-B1800
unix-B2400
unix-B4800
unix-B9600
unix-B19200
unix-B38400
unix-EXTA
unix-EXTB
unix-CSIZE
unix-CS5
unix-CS6
unix-CS7
unix-CS8
unix-CSTOPB
unix-CREAD
unix-PARENB
unix-PARODD
unix-HUPCL
unix-CLOCAL
unix-CBAUDEX
unix-B57600
unix-B115200
unix-B230400
unix-B460800
unix-B500000
unix-B576000
unix-B921600
unix-B1000000
unix-B1152000
unix-B1500000
unix-B2000000
unix-B2500000
unix-B3000000
unix-B3500000
unix-B4000000
unix-__MAX_BAUD
unix-CIBAUD
unix-CRTSCTS
unix-ISIG
unix-ICANON
unix-XCASE

unix-ECHO
unix-ECHOE
unix-ECHOK
unix-ECHONL
unix-NOFLSH
unix-TOSTOP
unix-ECHOCTL
unix-ECHOPRT
unix-ECHOKE
unix-FLUSHO
unix-PENDIN
unix-IEXTEN
unix-TCOOFF
unix-TCOON
unix-TCIOFF
unix-TCION
unix-TCIFLUSH
unix-TCOFLUSH
unix-TCIOFLUSH
unix-TCSANOW
unix-TCSADRAIN
unix-TCSAFLUSH
unix-SEEK_SET
unix-SEEK_CUR
unix-SEEK_END

Flags defined in the C include files. For any flag unix-X there is a #define unix-HAVE-X

unix-stat

Definitions for the size and field offsets, sizes and types for the C struct unix-stat

unix-sockaddr

Definitions for the size and field offsets, sizes and types for the C struct unix-sockaddr

unix-linger

Definitions for the size and field offsets, sizes and types for the C struct unix-linger

unix-sockaddr_in
unix-sockaddr_in-sin_family-offset
unix-sockaddr_in-sin_family-size
unix-sockaddr_in-sin_family-type
(unix-sockaddr_in-get-sin_family

Definitions for the size and field offsets, sizes and types for the C struct unix-sockaddr_in

unix-termios
unix-termios-c_iflag-offset

unix-termios-c_iflag-size
unix-termios-c_iflag-type
(unix-termios-get-c_iflag

Definitions for the size and field offsets, sizes and types for the C struct unix-termios

14.4 AVL-trees

AVL-trees are an efficient balanced binary tree.

You probably want to use the dict routines instead. They wrap around the AVL routines to provide a nicer interface.

(avl-get *tree cmpfun key default-value*)

Perform a search in the given tree using the given comparison function. Return the value stored for the given key, or 'default-value' if not found. Prefer the built-in 'default-avl-get' for performance reasons if the default comparison function suffices.

(avl-put *tree cmpfun key value*)

Perform insertion/replacement in the given tree using the default comparison function. Return the new tree. Prefer the built-in 'default-avl-put' for performance reasons if the default comparison function suffices.

(avl-fold *tree operation zero*)

Fold the given operation over all key-value pairs in the AVL-tree. The operation takes three arguments: the key and value found in the AVL-tree and the previously returned value of the operation.

(avl-fold-update *tree operation zero*)

Similar to avl-fold, but the operation can update the values in the tree. The operation returns a cons cell whose car is the new value and cdr is the return value of the operation. The entire fold-update returns a cons whose car is the tree with updated values and cdr the value of the last operation.

14.5 A Python-like dictionary

In Python, a dictionary maps a key (of some suitable type) to a value (of any type). In other languages, it might be called a hash table, hash map.

The functions in this section implement a fairly efficient dictionary using AVL tree routines (but this implementation detail is invisible to the caller). The key **MUST** be an integer, a symbol, or a string.

(dict-set *dict key value*)

Add a new value to the dictionary, replacing any old one for **KEY**. Return a new dict.

(dict-get *dict key*)

Get the value corresponding to a key in a dictionary. Return nil if the key had no value.

(dict-remove *dict key*)

Remove a value corresponding to a key. Return new dictionary.

(dict-set-from-list *dict pairs*)

Set many key-value pairs in a dictionary at once. The pairs are given as a list.

(dict-set-many *dict ... pairs*)

Set many key-value pairs in a dictionary at once. The pairs are given as separate arguments. This function is useful for initializing a dictionary with default values for many keys.

(dict-create ... *pairs*)

Create a new dictionary. If the variable argument list is empty, the new dictionary will also be empty. Otherwise the argument list shall contain pairs (cons cells) where the first element is a key and the second element is the value. The new dictionary will then contain these key-value pairs.

You can also use a plain nil value for an empty dictionary.

(dict-update *dict key fun*)

This is similar to dict-set, but the value to be set is done by calling '(fun old-value)'. Returns a new dict.

(dict-empty? *dict*)

If the dictionary empty?

(dict-smallest-key *dict*)

Return the smallest key in the dict, or nil if the dict is empty.

(dict-largest-key *dict*)

Return the largest key in the dict, or nil if the dict is empty.

(dict-get2 *dict1 key1 key2*)

Get a value from a dictionary within a dictionary. That is, 'dict1' at key 'key1' is a dictionary, and that one is indexed with 'key2'.

(dict-set2 *dict1 key1 key2 value*)

Set a value in a dictionary within a dictionary. See dict-get2.

14.6 Manipulating byte orders and integers in C structs as Lisp strings.

(c-swap-short *x*)

Swap the byte order of a 16-bit integer *x*.

(c-swap-int *x*)

Swap the byte order of a 32-bit integer *x*.

(c-int-to-string4 *v*)

Convert a 32-bit integer into a 4-character Lisp string in the host byte order.

(c-short-to-string2 *v*)

Convert a 16-bit integer into a 2-character Lisp string in the host byte order.

(c-string4-to-int *s*)

Read a 32-bit integer from a 4-byte (or longer) Lisp string in the host byte order.

(c-string2-to-short *s*)

Read a 16-bit integer from a 4-byte (or longer) Lisp string in big-endian, or network byte order.

(c-htons *?x*)

Equivalent of the C library function htons.

(c-ntohs *?x*)

Equivalent of the C library function ntohs.

(c-htonl *?x*)

Equivalent of the C library function htonl.

(c-ntohl *?x*)

Equivalent of the C library function ntohl.

(c-get-int *?string ?offset*)

Read a 32-bit integer from the given offset in the string.

(c-get-short *?string ?offset*)

Read a 16-bit integer from the given offset in the string.

(c-set-int *string offset value*)

Return a new string, identical to the given one, where the given 32-bit integer value has been placed in the given offset.

(c-set-short *string offset value*)

Return a new string, identical to the given one, where the given 16-bit integer value has been placed in the given offset.

14.7 IP

Functions and values that are useful for IP-programming.

ip-IPPORT_ECHO
ip-IPPORT_DISCARD
ip-IPPORT_SYSTAT
ip-IPPORT_DAYTIME
ip-IPPORT_NETSTAT
ip-IPPORT_FTP
ip-IPPORT_TELNET
ip-IPPORT_SSMTP
ip-IPPORT_TIMESERVER
ip-IPPORT_NAMESERVER
ip-IPPORT_WHOIS
ip-IPPORT_MTP
ip-IPPORT_TFTP
ip-IPPORT_RJE
ip-IPPORT_FINGER
ip-IPPORT_TTYLINK
ip-IPPORT_SUPDUP
ip-IPPORT_EXECSERVER
ip-IPPORT_LOGINSERVER
ip-IPPORT_CMDSERVER

TCP ports.

ip-IPPORT_EFSSERVER
ip-IPPORT_BIFFUDP
ip-IPPORT_WHOSERVER
ip-IPPORT_ROUTESERVER
ip-IPPORT_RESERVED
ip-IPPORT_USERRESERVED

UDP ports.

(ip-sockaddr_in *port ip*)

Make a Lisp-string that corresponds to the C struct sockaddr_in from the specified port and ip numbers.

14.8 List processing library functions

The functions in this section do things with lists. Lists are assumed to be constructed from cons cells in the usual manner.

(list ... *args*)

Return a list that contains all the values given as arguments.

(nth *list n*)

Return item number 'n' in a list or nil if item 'n' is outside the list. Counting starts at zero.

Arguments: 'list' the list of items 'n' index into the list

(nil? *list*)

Is 'list' equal to nil?

(accumulate *op so-far items*)

Accumulate a value by calling 'op' on each item in 'items'. The first call to 'op' gets 'so-far' and the first item as its arguments, successive calls get the previous call's return value and next item. The return value of the last call to 'op' is the return value of the function. If 'items' is empty, 'op' is never called and 'so-far' is returned.

(reverse *items*)

Reverse items in a list.

(append *a b*)

Return list containing elements of 'a' and then those of 'b'.

(map *func items*)

Call 'func' on each item, return list of return values.

(filter *pred? items*)

Return items for which 'pred?' returns true.

(for-each *func items*)

Call 'func' for each item. Return value of last call.

(len *list*)

Return number of items in 'list'.

(head *items count*)

Return the 'count' first items in a list.

(nth-cdr *items* *n*)

Skip 'n' items from the beginning of a list and return the tail. For example, (nth-cdr '(1 2 3) 2) would return (3).

(split *pred?* *items*)

Split a list into two: one with the items for which 'pred?' returns true, one with the rest. Return a pair containing the two lists.

(rotate *items* *steps*)

Rotate items in a list. For example, (rotate '(1 2 3) 1) would return (2 3 1).

(list-search *pred* *items*)

Return the first item in list for which 'pred' returns true, nil if none.

(sort *items* *less-than*)

Mergesort a list. 'less-than' is called to compare to elements in the list.

14.9 Math

This section contains functions that have to do with mathematics.

INT_MAX

INT_MIN

Constants for the largest and smallest numbers in our integer type.

(max *a ... args*)

Return the largest of its arguments (which must be integers).

(min *a ... args*)

Return the smallest of its arguments (which must be integers).

(abs *n*)

Return the absolute value of the argument, which must be an integer.

(unix-gettimeofday-subtract *stop* *start*)

Subtract two values returned by unix-gettimeofday, as usecs. Note that should the difference be over approximately four minutes, integer overflow will result.

14.10 Miscellaenous

This section contains functions that have no better section.

(require-version *required*)

Check that the version of the interpreter we're running under is compatible with the application. If not, panic.

(pr ... *args*)

Print out the values of the arguments and then a final newline. This differs from the built-in print function only in that the final newline is implicit.

(trace *trace?* *prefix values*)

If 'trace?' is true, write a log message, prefixed with 'prefix' and consisting of the values in 'values'. No space is printed between values.

14.11 Queues

The functions in this section implement simple functional queues. A queue is a data structure that can contain elements so that the first element added to the queue is the first element removed from the queue. First in, first out, that is. A functional queue is one that works without changing a data structure (it is impossible in Hedgehog, anyway). The queue, as implemented here, is $O(1)$ for adding elements, and $O(1)$ or $O(n)$ for removing elements. In a sequence of operations, the $O(n)$ is amortized so that the average operation is $O(1)$, even if there is an occasional $O(n)$.

(queue-make)

Create a new, empty queue.

(queue-empty? *queue*)

Is a queue empty?

(queue-length *queue*)

How many items are there in the queue?

(queue-add *queue item*)

Add an item to a queue. Return the new queue.

(queue-remove *queue*)

Remove the oldest item from a queue. Return a pair (item, new queue).

14.12 String operations

(int-byte *int n*)

Return the 'n'th byte of 'int' as a string. The least significant byte (bitmask 0xff) is 0, the next (0xff00) is 1, etc. Thus, (int-byte 0x12345678 0) will return (chr 0x78), and (int-byte 0x12345678 3) will return (chr 0x34).

(itob *i nbytes*)

Convert an integer 'i' into an 'nbytes' long string, in most significant byte first order. For example, (itob 0x1 2) returns "\0\2" and (itob 0x12345678 3) returns "\x34\x56\x78".

(btoi *str bytes*)

Convert a binary string to an integer. For example, (btoi "\x12\x34" 2) returns the value 0x1234. Note that the binary string is assumed to be in most-significant-byte first order.

(lsb-btoi *str pos num-bytes*)

Convert part of a binary string to an integer, least significant byte first. See 'btoi' for most significant byte first. Return the integer value.

(bcd-decode *str pos num-bytes*)

Decode a binary coded decimal (where every 4 bits is used to encode a decimal digit) from part of a string. Return the integer.

(strsplit *str sep*)

Split a string into parts at separators. For example, (strsplit "foo,bar,baz" ",") returns the list ("foo" "bar" "baz"). Note that the separator is searched from the end of the string towards the beginning. This matters for some patterns. For example, (strsplit "yaddafoofoofooyeehaa" "foofoo") results in the list ("yaddafoo" "yeehaa") and not the list ("yadda" "fooyeehaa").

(space? *c*)

Is 'c' a white space character (space, tab, CR, LF)?

(leading-spaces *str*)

Count the number of leading white space characters in 'str'.

(trailing-spaces *str*)

Count the number of trailing white space characters in 'str'.

(lstrip *str*)

Remove all leading white space characters in 'str'.

(rstrip *str*)

Remove all trailing white space characters in 'str'.

(strip *str*)

Remove all leading and trailing white space characters in 'str'.

(split-line1 *buffer*)

Split off the buffer into the first line, and everything but the first line. Return these as a pair. If there is no complete first line, return an empty string as the first line. The line terminator ("\n") is included in the first line, so that the caller may distinguish between an empty first line and no first line.

(begins-with? *str pat*)

Does a string begin with a patten (another string)?

(map-byte *func str*)

Call 'func' for each byte in a string, return a list containing the return values of the calls.

(split-bytes *str*)

Convert a string to a list of integers giving the values of the bytes in the string.

(strjoin *strings sep*)

Join a list of strings into one string, putting 'sep' between them.

(dehex *str*)

Convert a hexadecimal string (e.g., the output of 'hex') back to binary form. Non-hexadecimal digits in the string is silently discarded. A trailing single hexadecimal digit is also silently discarded.

(dehex-parse *str*)

Decode a hexadecimal string. Skip whitespace and other characters that are not hexadecimal digits. If the string is incomplete, i.e., contains an odd number of hexadecimal digits, the remaining part is returned. The return value is (cons decoded remaining).

(split-string-into-chunks *str chunk-size*)

Split a string into chunks that are of the same size, except possibly for the last one. Return a list containing the chunks.

14.13 State machines

Many applications can be written in terms of one or more finite state machines. This library provides a framework for implementing them.

Each state is a function that takes two arguments: the state machine's private data item (typically a dict, but could be anything) and either a message received by the machine, or nil if no message was received. The function has to be written as a sequence of variable bindings, goto "statements" indicating state change, wait "statements" indicating suspension of execution until the machine receives more input (or messages), and send "statements" which send messages to other machines. The goto and wait statements do not allow the execution to proceed to the next statement, but send does.

The machine may be woken up at any time before the desired timeout expires with the 'timeout message. When a state machine enters a new state (either from system start or after a goto), the machine receives an 'enter message. Readable and writable file descriptors are indicated with a message like '(readable ...) and '(writable ...) respectively.

The framework takes care of running the state machines, calling the proper functions and keeping track of the message passing between individual machines.

(sm-machine ?name ?state ?data ?readables ?writables ?timeout)

Create a state machine record instance. The arguments indicate the name of the state machine (an integer, symbol, or string), the (initial) state, the state machine's private data, lists of file descriptors the machine would like to read and write, and a maximum timeout before the framework sends a 'timeout message.

sm-max-timeout

The maximum timeout. Timeouts given to sm-machine larger than this value are silently capped to this value.

(sm-run-machines machines msgs)

Run the machines forever. The argument machines contains a dict of sm-machine records keyed by the name of the machine. The argument msgs is a queue of messages to be delivered.

(def-state (?state-name ?private-data ?msg)

A macro used to create the state function. The function takes two arguments, the private data which is associated to this particular instance of the machine and passed from state to state by sm-run-machines. The second argument, msg, is a message delivered by the framework to this state function. The body of the function is a sequence of either variable bindings as in let-statements, (send list-of-machines msg) -statements to send the given msgs to the given list of machines (nil if none, a single machine name is also ok), (goto new-state cond new-data) -statements to make a conditional transition to the a new state with a corresponding new version of the private data, and (wait cond new-data readables writables timeout) to stay in the same state until either one of the file descriptors in the readables-list has become readable, one of the file descriptors in the writables-list has become writable, or until a specified timeout has occurred.