Visible Standard Lisp User Manual

Arthur Norman

February 15, 2012

There is a separate larger document that explaims what `vsl`is and how it is implemented. This merely contains an explanation of how to fetch the sources and compile them, and a list of the functions that `vsl`then provides.

1 Fetching and building vsl

You can fetch `vsl`sources using `subversion`. If you do not have that installed on your computer already you need to discover how to fetch it. On Linux this will be easy using a package manager (typically `yum` on Fedora or `apt-get` on Debian or Ubuntu). On Windows if you visit `www.cygwin.com` you can fetch their setup program and install their environment – ensuring that you install `make`, `gcc` and `subversion`, and probably `tex`.

Then you can go

```
U=https://reduce-algebra.svn.sourceforge.net
V=$U/svnroot/reduce-algebra/trunk
svn co $V/vsl
```

which should create a directory called `vsl` and put a collection of files in it. The above instructions build up the path at `sourceforge` to fetch this from in parts so you only have to type a modest amount on each line. The files you will fetch amount to about a megabyte, and so should not be a severe strain on anything.

If you also want to use `vsl` to build a version of the Reduce algebra system you should follow up the subversion calls with another call (relying on the variable `$V` you set up to point to the subversion repository).

```
svn co $V/packages
```

This time you will end up with around 60 Mbytes in a directory called `packages`. That is the full set of sources for Reduce. Even though you are not liable to be able to make use of all of them the easiest recipe involves you fetching everything.

To build `vsl` and then try it you can then go

```
cd vsl
make
./vsl
(oblist)
(stop 0)
```

For the build to succeed you will need to have the `gcc` C compiler installed, and to rebuild the manual the `pdflatex`[1] command is required. Again you may need to use a package manager to ensure that they are available.

In the above small example you verify that `vsl` will run by calling the function `(oblist)` to display a list of all `vsl`'s built-in symbols, then you use `(stop 0)` to exit from the system.

To try Reduce (see `http://reduce-algebra/sourceforge.net` for full information. In particular there is a manual hidden there) you go

```
make reduce
./vsl
```

The "`make reduce`" step builds (much of) Reduce and saves the result in `vsl.img`. When you next start `vsl`it reloads that image file. Then you can try various algebraic examples. One of my favourites is

```
int(1/(x^6-1), x);
```

The `vsl`-hosted Reduce will be significantly slower than other versions and it certainly has severe limitations because `vsl` does not provide arbitrary prevision arithmetic. It is expected that there will be cases where some Reduce facilities try to use functions that `vsl` does not provide – and then crashes. It would be helpful if such cases can be collected and reported at least so that a section can be inserted here documenting Reduce limitations under `vsl`.

2 Summary of functions in `vsl`

`!$eof!$` *predefined variable*

When the `read` function (or its relatives) detect and end of file condition it returns this value of this variable. So a code fragment such as `(eq (setq x (read)) !$eof!$))` will read a Lisp expression, store it in a variable called `x` and test if it was in fact an end of file marker.

`!$eol!$` *predefined variable*

This value of this predefined variable is a newline character, so if you use

[1] typically instaled for you as part of some broader LaTeXpackage such as texlive.

`readch` it may be convenient to compare the result against this. See `blank`, `lpar` and `rpar` for other character values where it is convenient to have a name rather than needing to enter the (escaped) character directly.

`!*echo` *variable*

When Lisp reads in any input the variable `!*echo` is inspected. If its value is non-nil then the characters that are read get echoed to the current Lisp output stream. This is often useful when reading from a file. But often if you are accepting input in an interactive manner from the keyboard you would prefer `!*echo` to be `nil`. So `vsl` arranges that if it is started up with a file to read specified on its command line it makes `echo` default to `t`, while if no command-line arguments are given it defaults to `nil`. This often results in comfortable behaviour, but the user is free to set the variable explicity at any time to make things fit their needs.

`!,` *marker*

See the backquote entry.

`!,!@` *marker*

See the backquote entry.

`` ` `` *marker*

Lisp input can contain an ordinary Lisp expression preceeded by a backquote mark. Within the expression various sub-parts can be marked with either a comma, or a comma followed by an "at" sign. This notation is commonly used when defining Lisp macros. The effect is as if a longer segment of Lisp had been written to create a structures that is the same shape as the one given, but with the comma-introduced sub-parts expanded. This may be shown with an example. The form `` `(a ,b c ,(car d)) `` might behave like `(list 'a b 'c (car d))`. The rules for backquote do not guarantee exactly what expansion will be generated – just that when it is executed it will construct the required structure. In `vsl` the code will in fact use many calls to `cons` rather than the neat use of `list` shown here. An embedded "`,@`" within a backquoted expression is expected to stand for a list, whose values are spliced into the evantual result. This `` `(a ,@(car b) c) `` might expand as `(append '(a) (append (car b) '(c)))`. In many some Lisp systems the full expansion is performed while reading the input. In `vsl` the reader leaves backquote and comma markers in the structure it returns, and macros expand those when it is time to execute them.

`add1` *function 1 arg*

`(add1 n)` is merely a shorthand for `(plus n 1)`, in other words it adds one to its argument. It is often useful when counting. See alse `sub1`.

`and` *function n args*

In `vsl` `and` is implemented as a special form. If evaluates its arguments one at a time, and returns `nil` if one of them evaluates to `nil`. If none of them yield `nil` its value if the value of the final argument. If you interpret `nil` as *false* and anything else as *true* then this matches a simple understanding of the of an operation that could reasonably called `and`. See also `or`. A different way of explaining `and` would be to give an equivalence: `(and a b c)` could be replaced by `(if a (if b c nil) nil)`, with similar expansion for cases with larger or smaller number of arguments. In `vsl` `(and)` (ie without any arguments) yields `t`.

`append` *function 2 args*

If you have two lists then `append` can form their concatenation. So `(append '(a b) '(c d))` yields `(a b c d)`. The result will share structure with the second argument – a fact that only matters if you later use `rplaca` or `rplacd`. Some other Lisps may permit you to give `append` more than two arguments, and will then append all the lists into one, but `vsl` does not.

`apply` *function 2 args*

If you have either the name of a function or a lambda-expression (see `lambda` for more explanation) you can call it on a collection of arguments that are provided in a list. The function `apply` that does this is really there just because its capability has to be part of the Lisp interpreter. For instance since `cons` takes just two arguments you could invoke it by giving the symbol `cons` as `apply`'s first argument and a list of length two as its second: `(apply 'cons '(a b))`. This would return `(a . b)`. If the first argument to `apply` is a macro rather than an ordinary function this can be used to perform macro expansion. You should not try using `apply` on a special form (`fsubr`).

`assoc` *function 2 args*

An association list is a list of pairs, and each pair (`cons`) is though of as constisting of a key and a value. `assoc` searches an association list looking for a given key. If it find it then it returns the pair that contains it. Othewise it returns `nil`. Thus `(assoc 'b '((a . 1) (b . 2) (c . 3)))` will return `(b . 2)`. The test for keys is make using the `equal` function.

`atan` *special form*

The arctangent function, working in radians. See `sin` and `cos`.

atom *function 1 arg*

Any Lisp object that is not a list or a pair – that is to say that could not have been created by the `cons` function, is known as an *atom*. Thus symbols, numbers, strings and vectors are all atoms. The function `atom` checks its argument and returns `t` (for *true*) if it is atomic and `nil` otherwise.

blank *predefined variable*

A predefined variable whose value is the symbol whose name is a single space character. One could otherwise refer to that symbol by writing `'!␣`, but many people find writing the word `blank` makes things clearer because it does not involve having a significant but non-printing character.

caaar *function 1 arg*

A range of names of the form `c`*xxx*`r` with the intermediate letters being either `a` or `d` are provided as functions that are merely combinations of uses of `car` and `cdr`. Thus `(caaar x)` means just the same as `(car (car (car x)))`, and `(caddr x)` means `(car (cdr (cdr x)))`. In `vsl` these are provided for up to three intermediate letters.

caadr *function 1 arg*

See `caaar`.

caar *function 1 arg*

See `caaar`.

cadar *function 1 arg*

See `caaar`.

caddr *function 1 arg*

See `caaar`. Can be though of as returning the third item in a list.

cadr *function 1 arg*

See `caaar`. Can be though of as returning the second item in a list.

car *function 1 arg*

Data structures in Lisp are made up with the various sorts of atom (symbols, numbers, strings and so on) as basic elements and with `cons` cells used to build them up into potentially large and complicated lists and trees. If you think of a structure as a list then `car` extracts its first element (and `cdr` its tail). If you think of it as a binary tree then `car` gets the left part and `cdr` the right. The basic identity is that `(car (cons a b)) = a`. See also `cdr`.

`cdaar` *function 1 arg*
See `caaar`.

`cdadr` *function 1 arg*
See `caaar`.

`cdar` *function 1 arg*
See `caaar`.

`cddar` *function 1 arg*
See `caaar`.

`cdddr` *function 1 arg*
See `caaar`.

`cddr` *function 1 arg*
See `caaar`.

`cdr` *function 1 arg*
`(cdr (cons a b))` = `b`. See `car`.

`char!-code` *function 1 arg*
If you have a symbol or a string denoting a single character then the function `char!-code` will return a numeric code for it. The code used in `vsl` is ASCII so for instance the code for a blank character is 32, that for the digit "0" is 46 and a capital "A" is 65. See `code!-char` for conversion in the other direction.

`close` *function 1 arg*
When an input file has been opened for reading or writing you should use `close` it once finished with it. This is especially important for output files because it could be that some material will remain buffered and so will not be written until the file is closed. See `open`.

`code!-char` *function 1 arg*
`(code!-char 97)` would return a symbol whose name is the character with code 97 (in this case a lower case "a"). Similarly for other codes typically in the range 0 to 255.

`compress` *function 1 arg*
If you have a list of identifiers or strings then `compress` treats each as standing for its first character and returns the Lisp expression you would get if ytou read from a document that contained those characters. This would normally be used when each item in the input list was just a single character.

Because in `vsl` the `compress` function is implemented by just involing `read` with input redirected from the list you can create symbols, numbers, strings and even lists this way. See `explode`.

`cond` ***special form***

The primitive comditional operator in Lisp is `cond`. It is a special form (i.e. it does not evaluate its arguments in the standard way. Its use is as in

```
(cond
   (p1 e1a e1b e1c ...)
   (p2 e2a e2b ...)
   ...
```

where `p1`, `p2` etc. are predicates, and the sequence of expressions (for instance `e1a...`) that follow the first one that yields a non-`nil` value are computed. The result returned is the final thing that `cond` evaluates. There are many examples of uses of `cond` in the sample code here. Some people prefer to use `if` or `when` instead, but at least historically `cond` came first.

`cons` ***function 2 args***

Lisp data is based on lists and trees, and `cons` is the key function for creating them. The term `cons`-cell is used for the object that the function creates. Such a cell has two componente, is `car` and its `cdr`. The apparently strange names for these related to the atchitecture of an early computer on which Lisp was first developed. If you have a list structure `l` and an item `a` then `(cons a l)` is a list just one element longer than `l` was formed by putting `a` in front of the original. In Lisp it is much more expensive to attach a new item to the tail of a list. To do that would typically involve `(append l (list a))` and especially if `l` was long could be slow. So in Lisp it is normal to build up lists by succesively addint items to the head using `cons`. See also `car`, `cdr` and `list`.

`copy` ***function 1 arg***

It is normally only necessary (or indeed useful) to make a copy of a Lisp structure if you are then about to use destructive operations such as `rplaca` on the original, but this function is provided in case you do ever need to.

`cos` ***function 1 arg***

It would be easy to extend the `vsl` implementation to provide a full set of mathematical functions, but to keep things small the initial version only provides a few key cases: `sin`, `cos`, `atan`, `exp`, `log` and `sqrt`. Each of these can be given either an integer or floating point argument but they always return a floating point result. The trigonometric functions work in radians rather than degrees.

de *special form*

To define a new function evaluate `(de name arglist body)`, the special form that is provided for this purpose. After you have defined something you could retrieve the definition you had set up using `(getd 'name)`.

deflist *function 2 args*

Sometimes when setting up data you need to perform a succession of `put` operations all using the same property name. `deflist` provides a shortcut so you can write something like

```
(deflist
   '((a A)
     (b B)
     (c C))
   'propname)
```

and have the same effect as

```
(put 'a 'propname 'A)
(put 'b 'propname 'B)
(put 'c 'propname 'C)
```

df *special form*

`df` is rather like `de` except that it allows you to defined a new special form. A special form must be defined as if it has just one argument, and this will receive the whole of the "argument" information from any call without any evaluation having happened. Often the body of a special form will thus wish to use `eval` to make evaluation happen. In most Lisp programs it will be unusual to introduce new special forms. See also `dm` for an alternative way (also sometimes controversial) for extending the syntax of Lisp.

difference *function 2 args*

Subtract one number from another. If either value is floating point the result will be floating point.

divide *function 2 args*

Divide one integer from another and return the `cons` of the quotient and remainder. The idea behind this function was that when integers are divided it is common to require both quotient and remainder, so having a one function to return both might be helpful. In Lisp systems that support very high precision arithmetic this can indeed save time. In `vsl` you will probably do as well calling `quotient` and then `remainder`.

dm *special form*

A Lisp Macro is something that when evaluated produces futher executable Lisp to be its replacement. The special form introduced by `dm` can defined a new one. In general it will be sensible to define macros before you define any functions that use them. Some people believe that extensive use of macros can make your code harder to read and debug, and so would rather you did not use them at all.

do *macro*

A perhaps over-general form of loop can be specified by the `do` macro or its close cousin `do!*`. The structure of aan invocation of it is

```
(do ((var1 init1 step1)
      (var2 init2 step2)
      ..)
     (endtest result ...)
     body
     ...)
```

and a concrete exammple is

```
(do ((x 10 (add1 x)))
    ((greaterp x 20) 'done)
    (print (list x (times x x))))
```

The difference between `do` and `do!*` is that the former processes all its initialisation and update of variables in parallel, while the latter acts sequentially. This is similar to the relationship between `let` and `let!*`.

do!* *macro*

See `do`.

dolist *macro*

Simple iteration over a lisp can be performed using the `dolist` macro, where a typical tiny example might be

```
(dolist (x '(1 2 3) 'result) (print x))
```

which prints the numbers 1, 2 and 3 and then returns the value `result`. In many cases you will merely omit the result part of the expression and then `dolist` will return `nil`.

dollar ***predefined variable***

A predefined variable whose value is the symbol whose name is a dollar character. See `blank` for another similarly predefined name.

dotimes ***macro***

Counting is easy with `dotimes`. It starts from zero, so

```
(dotimes (x 5 'result) (print x))
```

will print values 0, 1, 2, 3 and 4 before returning `result`.

eq ***function 2 args***

If you wish to test two Lisp items for absolute identity then `eq` is the function to use. If you enter the same spelling for a symbol twice Lisp arranges that you get the same symbol, but it is possible – even probable – that strings or large numbers can fail to be `eq` even if they look the same. Two list structures are `eq` only if their top-level `cons` cells are identical in the sense that even if you received them via different paths they are the output from the same call to `cons`. See `equal` for a more expensive but perhaps more generous equality test.

eqcar ***function 2 args***

`(excar a b)` is like `(and (not (atom a)) (eq (car a) b))`.

equal ***function 2 args***

Wile `eq` compares objects for absolute identity, `equal` compares them to see if they have the same structure. `equal` understands how to compare big and floating point numbers, strings and vectors as well as lists. To illustrate the difference between the two functions consider

```
(setq a (cons 1 1000000000))
(setq b (cons 1 1000000000))
(eq a b)
(equal a b)
(eq (cdr a) (cdr b))
(equal (cdr a) (cdr b))
```

In each case `eq` returns `nil` wiile `equal` will return `t`. Although you should not in general rely on `eq` when comparing numbers, in `vsl` all small numbers are represented in a way that will allow `eq` to handle them reliably. If `vsl` is runing on a 32-bit system the range is -268435456 to 268435455. If the system had been built for a 64-bit machine it is much larger.

error ***function 2 args***

If a user wants to report that something has gone wrong it can call the `error` function. This is given two arguments, and they will be displayed in any message that is printed. See `errorset` for information about how to control the amount of information displayed when an error occurs.

errorset ***function 3 args***

The default situation is that when `vsl` encounters and error it unwinds from whatever it was doing and awaits the next item of input from the user. The function `errorset` can be used to trap errors so that a program can respond or continue in its own way. It can also control how much diagnostic output is generated. A call `(errorset form msg trace)` will evaluate the Lisp expression `form` rather in the way that `eval` would have. If there is no error it returns a list of length one whose element is the value that was computed. If the evaluation of `form` failed then `errorset` returns an atom rather than a list, so its caller can be aware of the situation. The argument `msg` can be non-`nil` to indicate that a short (typically one line) report is displayed on any error. If `trace` is non-nil then a report showing the nesting of function calls will also be shown. If both arguments are `nil` then the error and recovery from it should be silent. See `eval` for a sample use.

eval ***function 1 arg***

An approximation to how Lisp interacts with the user is

```
(while t
    (errorset '(print (eval (read))) t t))
```

where `eval` takes whatever form was read and evaluates it. The `eval` function (and its relative `apply`) can be used anywhere in Lisp code where a data structure needs to be interpreted as a bit of Lisp code and obeyed.

exp ***special form***

The exponential function. See `log`.

expand ***function 2 args***

In some Lisp implementations it would be useful (for instance for efficiency) to transform some uses of functions taking an indefinite number of arguments (for instance `plus`) into sequences of calls to versions taking just two arguments. The `expand` function is intended to help with this. Its first argument is a list of expressions, the second a (two argument) function to be used to combine them. For instance `(expand '(a b c)` 'plus2) will yield (plus2 a (plus2 b c)). This could be useful if plus were to have been implemented as a macro expanding to multiple uses of plus2 rather than as a special form, and if vsl provided the two-argument function concerned (which at present it does not!).

explode *function 1 arg*

Any Lisp item can be processes as by prin but with the resulting sequence of characters being collected as a list rather than being printed directly. explode does this, while explodec behaves like princ. So (explode '("a" . 3)) will be (!(!" a !" ! !. ! !3 !)). explode can be useful to find the sequence of letters making up the name of a symbol (or just to make it possible to see how many there are).

explodec *function 1 arg*

See explode

expr *symbol*

The function getd can retrieve the function definition (if any) associated with a symbol. The value returned is (type . value) where the type is one of the symbols expr, subr, fexpr, fsubr or macrp. The case expr indicates that the function is a normal-style function that has been defined in Lisp. The value information following it in the result of getd is the Lisp structure representing it. fexpr is for special forms defined in Lisp (using df. subr and fsubr and ordinary and special functions that have been defined at a lower level than Lisp (in other words things that form part ofthe vsl kernel). macro marks a macro as defined using dm. With the default vsl image (getd 'caar) returns (expr lambda (x) (car (car x))).

f *predefined variable*

f is a variable pre-set to have the value nil. This exists because at one stage people tended to want to use t for *true* and f for *false*. In most cases it will be safer to use nil directly if that is what you mean, and attempts to use f as a definitive denotation of *false* cause trouble when you try using the name as an ordinary variable, as in (de fff (a b c d e f g) ...).

fexpr *symbol*

See expr.

fix *function 1 arg*

If you have a floating point number and want convert it to an integer you can use the function `fix`. It truncates the value towards zero while doing the conversion.

`fixp` *function 1 arg*

The predicate `fixp` tests if its argument is an integer, and if it is it returns `t`. See also `numberp` and `floatp`. You are permitted to test any Lisp object using `fixp` and note that `(fixp 2.0)` will be `nil` because 2.0 is a floating point number even if its value is an integer.

`float` *function 1 arg*

Converts from an integer to a floating point number.

`floatp` *function 1 arg*

Test if an object is a floating point number. See `fixp`.

`fsubr` *symbol*

See `expr`.

`gensym` *function 0 args*

Sometime in a Lisp program you need a new sybol. One that is certain not to clash with any others you may have used already. `(gensym)` will create a fresh symbol for you. Such symbols should be though of as being anonymous. In `vsl` they do not even have a name unless and until you print them. At that stage a name will be allocated, and it will be of one from the sequence `g001`, `g002`,`...` However if you type in the characters `g001` that will not refer to the generated symbol that was displayed that way – you will get an ordinary symbol that you may confuse with the gensym but that Lisp will not.

`geq` *function 2 args*

This is a predicate that returns `t` if its first argument is greater than or equal to its second. Both arguments must be numbers. See `leq`, `greaterp` and `lessp`.

`get` *function 2 args*

Every symbol has a property-list, which can be retrieved directly using `plist`. The main functions for saving and retrieving information on property lists are `put` and `get`. After you have gone `(put 'name 'tag 'value)` a call `(get 'name 'tag)` will return `value`. See also `remprop`. An extended version of the library could define functions `flag`, `flagp` and `remflag` to store flags rather then more general properties, but `vsl` only supplies the basics.

getd *function 1 arg*

See expr.

gethash *function 2 args*

If h is a hash table then (gethash 'key h) retrieves the value stored in it under the indicated key by some previous call to (puthash 'key h 'value). See mkhash, puthash and remhash.

getv *function 2 args*

If tx v is a vector then (getv v n) returns the nth element ot if. See mkvect, putv and upbv.

go *special form*

See prog.

greaterp *function 2 args*

(greaterp x y) is true if x and y are numbers with x larger then y. See geq, lessp and leq.

if *macro*

The fundamental conditional form in Lisp is cons, but for convenience the macros if amd when are supplied. if takes two or three arguments. The first is a predicate – the condition that is to be tested. The next is the value to return, while the last is the result required if the condition is *false* and defaults to nil. when also takes a predicate, but all its further arguments are things to be obeyed in sequence if the condition holds. Thus (when p a b c) behaves like (if p (progn a b c) nil).

input *symbol*

See open.

lambda *symbol*

Some people will believe that lambda is the key symbol standing for the essence of Lisp. Others view it as a slight curiosity mostly of interest to obsessive specialists. It introduces a notation for a function that does not require that the function be given a name. This is inspired by Alonzo Chruch's λ-calculus. The denotation of a function is a list with lambda as its first element, then a list of its formal parameters, and finally a sequence of values that are to be evaluated when the function is invoked. So (lambda (x) (plus x 1)) is a function that adds one to its argument. If you try writing lambda expressions with bodies that refer to variables other than their formal arguments then you will need to read and understand the secion of this book that discusses deep and shallow binding strategies in an implementation of Lisp. This issue in fact arises with named functions defined using `de` as well.

`last` *function 1 arg*

If you have a list then `last` can return the final element in it. Recall that the first item in a list can be extracted using `car`, and note that `last` is goig to be slower, so where possible arrange what you do so that you access the start of your lists more often than their ends.

`lastcar` *function 1 arg*

This function is just like `last` except that if `last` is called on an empty list if reports an error, while `lastcar` merely returns nil.

`leftshift` *function 2 args*

Take an integer value, treat it as a bit-pattern and shift that leftwards by the specified amount. Return the result as an integer. Generally `(leftshift x` n`)` has the same effect as multiplying `x` by 2^n. See `rightshift`.

`length` *function 1 arg*

This function returns the length of a list. If its argument is `nil` it returns 0.

`leq` *function 2 args*

A test for "less than or equal". See `geq`, `greaterp` and `lessp`.

`lessp` *function 2 args*

A test for "less then". See `geq`, `greaterp` and `leq`.

`let` *macro*

Sometimes it is convenient to introduce a new name for some result you have just computed and are about to use. The `let` macro provides a way to do this. So as an example where four temporary values are being introduced, consider

```
(let ((u (plus x y))
      (v (difference x y))
      (xx (times x x))
      (yy (times y y)))
   (list (difference xx yy)
         (times u v)))
```

In really old fashioned Lisp this would have been achieved using `prog` as in

```
(prog (u v xx yy)
    (setq u (plus x y))
    (setq v (difference x y))
    ...
    (return (list (difference xx yy) ...)))
```

and yet another scheme would use an explicit lambda-expression

```
((lambda (u v xx yy)
    (list (difference xx yy) (times u v)))
 (plus x y)
 (difference x y)
 (times x x)
 (times y y))
```

Of all these the version using `let` is liable to be the clearest and neatest. Actually the versions using `prog` and `lambda` can have different behaviours sometimes. `let` and `lambda` introduce all their new variable simultaneously, and so the definition given for a later one can not depend on an earlier one. `let!*` is like `let` but introduces one variable at a time so that subsequent ones can depend on it, and that is closer to how the naive use of `prog` would work. Thus

```
(let!* ((x2 (times x x))
        (x4 (times x2 x2))
        (x8 (times x4 x4)))
   (times x x4 x8))
```

returns the thirteenth power of `x`, while if `let` had been used rather than `let!*` you would have received an error message about `x2` being undefined that arose when it was used to define `x4`.

let!* *macro*

ASee `let`.

lispsystem!* *predefined variable*

It is sometimes useful to allow Lisp code to adapt based on knowing something about the particular Lisp implementation it is running on. In Standard Lisp (and hence `vsl`) environment information is provided in a predefined variables called `lispsystem!*`. In `vsl` the only information put there is a symbol `vsl` to identify the Lisp system in use. But it would be easy to extent the code in `vsl.c` to put in whatever extra information about the host computer anybody felt might be relevant.

list ***special form***

The fundamental function for building Lisp data-structures is `cons`, but by convention a list is a chain of `cons` cells ending with `nil`. Created in the fundamental manner a list of length 4 might be built using `(cons 'a (cons 'b (cons c (cons 'd nil))))`. That is correct but clumsy!. The special form `list` accepts an arbitrary number of arguments and forms a list out of them: `(list 'a 'b 'c 'd)`. It will therefore be common to use one call to `list` in place of multiple uses of `cons` whenever possible.

list!* ***special form***

The structures created using `list` are always automatically provided with a `nil` termination. Sometimes a list-like structure is wanted with some other end. This may arise for instance when putting multiple items onto the front of an existing list. The function `list!*` can achieve this, and taking an example that puts 4 items ahead of the termination, the long-winded form `(cons 'a (cons 'b (cons c (cons 'd 'e))))` could be replaced by the much more concise `(list!* 'a 'b 'c 'd 'e)`. As a special case `list!*` with just two arguments degenerates to being exactly the same as `cons`.

log ***special form***

The (natural) logarithm of a value. See `exp`.

logand ***special form***

If integers in Lisp are represented in binary form (and for those who want the full story, negative values in two's complement) then a number can be thought of as a string of bits. `logand` takes an arbitrary number of integers and performs an logical "and" operation on each bit position, so that a "1" is present in the output only all of the inputs have a "1" in that position. The result is returned as an integer. See also `logor`, `lognot` and `logxor` for operations, and `leftshift` and `rightshift` for re-aligning bits.

lognot ***function 1 arg***

See `logand`. This function negates each bit.

logor ***special form***

See `logand`. This function yields a "1" when any input has a "1" in that place.

logxor ***special form***

See `logand`. This yields a "1" if an odd number of inputs have a "1" in the cooresponding place, and is "exclusive or".

lpar ***predefined variable***

A predefined symbol whose value is the symbole whose name is a left parenthesis. See also `rpar`.

macro ***symbol***

See `expr`.

map ***function 2 args***

There are a number of functions whose names begin with `map`. These take two arguments, on a list and the second a function. Each of them traverses the list calling the given function for each position on it. `map`, `maplist` and `mapcon` each pass first the original list and then each succesive tail of it. `mapc`, `mapcar` and `mapcan` pass the successive items in the list.

In each case the three variants do different things with the results returned by the function. `map` and `mapc` ignore it and in the end just return `nil`. This is only useful if the function that is provided has side-effects. For instance it might print something. `maplist` and `mapcar` build a new list out of the results, and so theior overall result is a list the same length as the input one. Finally `mapcon` and `mapcan` expect the function to return `nil` or some list, and they use `nconc` to concatenate all those lists.

Many people find `mapc` is the most useful, but then that the `dotimes` macro (which achieves a similar effect) is easier to use: compare

```
(setq a '(1 2 3 4))
(mapc a '(lambda (x) (print (times x x))))
(dolist (x a) (print (times x x)))
```

which achieve similar effects.

mapc ***function 2 args***

See `map`.

mapcan ***function 2 args***

See `map`.

mapcar ***function 2 args***

See `map`.

mapcon ***function 2 args***

See `map`.

maplist ***function 2 args***

See `map`.

minus ***function 1 arg***

Negates a number.

minusp ***function 1 arg***

Tests if a value is a negative number. Note that in `vsl` it is legal to call `minus` with an argument that is not even a number, in which case it will return `nil` to indicate that it is not negative, but in many other Lisp dialects you should only give `minusp` numeric input.

mkhash ***function 1 arg***

The hash-table capability built into `vsl` uses `(mkhash n)` to create a table, `puthash` to insert data and `gethash` to retrieve it. `remhash` can remove data. The argument to `mkhash` is used to control the size of the table, and might reasonable by chosen to be a fifth or a tenth of the number of keys you expect to store. Searches within hash tables are based on `eq`-style identity and are expected to be faster than various alternative (if simpler) schemes that could be considered.

mkvect ***function 1 arg***

In `vsl` a call `(mkvect` n`)` creates a vector where subsequent uses of `putv` and `getv` can use index values in the range from 0 to n. This results in the vector having $n + 1$ elements. A whole vector counts as an atom in Lisp, not as a list. If `v` is a vector then `(upbv v)` returns its upper bound – the largest subscript legal for use with it.

nconc ***function 2 args***

Given two lists, `nconc` concatenates them using a `rplacd` on the final cell of the first list. This avoids some extra storage allocation that `append` would have had to make, but overwrites part of the first list, and unless used with sensitivity that can cause trouble.

neq ***function 2 args***

`(neq a b)` yields exactly the same result as `(not (equal a b))`.

nil ***predefined variable***

The symbol `nil` is used in Lisp to denote an empty list, or to mark the end of a non-empty one. It is used to mean *false*, with anything non-nil being treated as *true*. The value of `nil` is `nil`. In some Lisp systems (but not this one or its close relatives) it is arranged that `car` and `cdr` may accept `nil` as an argument and yield `nil`. Here you are not allowed to do that so `(cdr nil)` will report an error just as `(cdr 'any_other_atom)` would.

`not` ***function 1 arg***

When a value is being thought of as a truth-value the function `not` can be used to invert it. Because *false* is represented by `nil` it turns out that `not` behaves identically to `null`.

`null` ***function 1 arg***

Tests if its argument is `nil`. Often used to detect when a list is empty.

`numberp` ***function 1 arg***

Returns `t` if its argument is a number. See also `fixp` and `floatp` that check for specific sub-categories of numbers.

`oblist` ***function 0 args***

The term "object list" is historically used in Lisp to refer to the symbol table that keeps track of all the identifiers that are in use. Its purpose is to arrange that every time you enter the same sequence of characters you get the same symbol back. The function `(oblist)` returns a list of all symbols in this table. This table of symbols important to Lisp was started by taking the output from `oblist` and sorting and formatting it. The number of symbols in the object list gives some idea of the size of the Lisp implementation. With `vsl` there are a couple of hundred symbols known before you start adding more. With the C and the Java coded implementations of Standard Lisp used with the Reduce algebra system there are around four times as many.

`onep` ***function 1 arg***

Test it its argument is 1 or 1.0. See `zerop`.

`open` ***function 2 args***

To access data in a file you first `open` the file. A file can be opened either for reading or writing, as in `(open "input.dat" 'input)` or `(open "newfile.out", 'output)`. In each case `open` returns an object that can be passed to `rds` or `wrs` to select that stream for use. When finished with any file that has bene open should be tidied up by handing its descriptor to the `close` function. As well as providing access to files, `open` can be used to launch another program, with Lisp output to the associated stream made available to that program as its standard input. This is done using the construction `(open "programname" 'pipe)`.

`or` ***special form***

See `and`.

`output` ***symbol***

See `open`.

pipe *symbol*

See `open`.

plist *function 1 arg*

Every symbol has a property-list and the `plist` function returns it. Normally this will only be used as a matter of interest, since `put` and `get` are the proper functions for storing and retrieving information from property lists.

plus *special form*

Adds an arbitrary number of values. If any one of them is floating point the result will be floating point.

preserve *function 0 or 1 arg*

If you call `preserve` a copy of the current status of everything in your Lisp world is written to a file called `vsl.img`. When `vsl` next starts it reloads this file (unless the `-z` command line option is used). This capability can be used to build an image file containing all the definitions and settings that make up a program so that when `vsl` is started they are all immediately available.

prettyprint *function 1 arg*

The ordinary print functions in `vsl` fit as much on a line as they can. In contrast `prettyprint` attempts to make its output more human-readable by indenting everything in a systematic manner. Su of you want to print out some Lisp code in a format where it is easier to read it may be useful.

prin *function 1 arg*

The family of print functions supplied with `vsl` consists of `prin`, `princ`, `print` and `printc`. The basic behaviour of each is that they print their argument to the current output stream. The versions with a "`c`" omit any escape marks, and when printing strings do not print quote marks, and so the output is perhaps nice for a human reader but could not be presented back to Lisp. The ones without a "`c`" insert escapes (exclamation marks) in names that include characters other than letters and digits, and do put quote marks around strings. The versions with a "`t`" terminate the output line after displaying their argument, so that a sequence of calls to `prin` display all the values on one line, while `print` puts each Lisp form on a separate line. See `terpri` and `wrs`.

princ *function 1 arg*

See `prin`.

print ***function 1 arg***

See `prin`.

printc ***function 1 arg***

See `prin`.

prog ***special form***

`prog` feels like an archaic feature inherited from the early days of Lisp, amd provides a range of capabilities. Firstly it itroduces some local variables, then it allows for the sequential execution of a sequence of Lisp forms, with a labels and a `go` statement to provide control. Finally a `prog` block only returns a non-nil value if the `return` function is called within it to provide one. Here is an illustration of the use of these to compute and print some Fibonacci numbers and then return the symbol `finished`

```
(prog (a b n c)
   (setq a 0 b 1 n 0)
top(when (greaterp n 10) (return 'finished))
   (setq c b b (print (plus a b)) a c n (add1 n))
   (go top))
```

progn ***special form***

There are a number of contexts in Lisp where you can write a sequence of expressions and these are evaluated in turn with the result of the final one as the overall result. `progn` can provide this capability in any other situation where it is useful.

psetq ***macro***

See tx setq, but `psetq` arranges to evaluate all the new values before updating any of the variables. A typical use of it would be to exchange the values in two variables, as in `(psetq a b b a)`.

put ***function 3 args***

See `get` and `deflist`.

puthash ***function 3 args***

See `mkhash`, `gethash` and `remhash`.

putv ***function 3 args***

See `mkvect` and `getvec`.

quote ***special form***

Normally each sub-part of a Lisp program will be evaluated – that is to say treated as program not data. The special form `quote` protects its argument from evaluation and so is used when you wish to specify data. This is so common and so important that Lisp provides special syntax for it. A Lisp expression preceeded by a single quote mark `'` is expanded into an application of the `quote` function. Thus `'(a b c)` means exactly the sane as `(quote (a b c))`.

quotient *function 2 args*

Form the quotient of two numeric arguments.

rassoc *function 2 args*

`rassoc` is just like `assoc` except that it looks for a match against the second component of one of the pairs in a list rather than the first. So it is a sort of reversed-`assoc`. Thus `(rassoc 2 '((a . 1) (b . 2) (c . 3)))` returns `(b . 2)`.

rdf *function 1 arg*

`rdf` reads and interprets all the Lisp code in the file whose name it is given as an argument.

rds *function 1 arg*

To read data from a file you first open the file (using `open`) then select it as the current input stream using `rds`. A call to `rds` returns the previously selected input stream, and very often you will want to save that so you can restore it later. A reasonably complete (and slightly cautious) example would be

```
(let* ((instream (open "filename" 'input))
       (oldstream (rds instream)))
   (errorset '(process (read)) t t)
   (rds oldstream)
   (close instream))
```

This shows saving the existing input stream and restoring it at the end. It uses `errorset` to ensure that this happens even if processing the input from the file fails.

read *function 0 args*

This reads a full Lisp object from the current input stream (which is by default from the keyboard, but can be changed using `rds`). The item can be a symbol, a number, a string or a list. It can also start with a quote mark or backquote. The function `read` is the one that is normally used when Lisp wants to grab input from the user, so the standard Lisp top level behaviour is as if it were obeying `(while t (print (eval (read))))`. If `read` finds the end of an input file it returns `!`eof!.

`readch` *function 0 args*

This reads a single character, returning a Lisp symbol that has that character as its name. If `readch` finds the end of an input file it returns `!`eof!.

`readline` *function 0 args*

This reads a whole line and returns a symbol made up from the characters found. If `readlione` finds the end of an input file it returns `!`eof!.

`remainder` *function 2 args*

This divides a pair of integers and returns the remainder.

`remhash` *function 2 args*

`(remhash 'key table)` removes a hash table entry previously inserted by `puthash`.

`remprop` *function 2 args*

`(remprop 'symbol 'indicator)` removes a property previously set up using `put`.

`return` *function 1 arg*

This is used with `prog`.

`reverse` *function 1 arg*

The ordinary function for reversing a list.

`reversip` *function 1 arg*

A function that reverses a list in a way that re-uses the existing `cons`-cells so as to avoid any need to allocate fresh ones. This necessarily destroys the input list by overwriting parts of it (using `rplacd`) so should only be used when you are certain that nobody else needs that list.

`rightshift` *function 2 args*

See `logand` and friends. This shifts the bits in a number right, and at least for positive values `(rightshift a` n`)` has the same effect as dividing `a` by 2^n.

`rpar` *predefined variable*

A predefined symbol whose value is the symbole whose name is a right parenthesis. See also `lpar`.

`rplaca` *function 2 args*

If you have a `cons`-cell you can use `rplaca` to replace the `car` field, `rplacd` to replace the `cdr` field or `rplacw` to replace both. In each case the result of the function is the `cons` cell that has been updated. Use of these functions can corrupt existing structures and create cyclic ones that lead to all sorts of trouble, and so they should only be used when there is a compelling reason to need a side-effect.

rplacd ***function 2 args***

See `rplaca`.

rplacw ***function 2 args***

See `rplaca`.

sassoc ***function 3 args***

Here we have another variant on `assoc`. In fact `sassoc` is like `assoc` except that it has an extra argument, and if the key it is looking for is not found in the association list rather then returning a simple value `nil` it returns the result from calling that final argument as a function with no arguments. While this has a long history of being part of Lisp I suspect that very few people use it these days.

set ***function 2 args***

This behaves like `setq` except that rather than being a special form it takes exactly two arguments, and treats the first as the name of a variable and the second as a value to store into that variable. It is not very common to need to make the name of a variable that you are assigning to a computed value in this manner.

setq ***special form***

When you have a Lisp variable you can change its value using `setq`. In fact `setq` allows you to make several updates, one after another, in one call. Its arguments alternate between being variable names and expressions to compute values to set them to. So for instance `(setq a 1 b 2)` sets `a` to `1` and `b` to `2`. See `psetq` for a variant that does all the assignments in parallel.

sin ***function 1 arg***

This is the usual trigonometic function, accepting its argument in radianc. See `cos` and `sqrt`.

spaces ***function 1 arg***

`(spaces` n`)` prints n blanks.

sqrt ***function 1 arg***

Computes the square root of a number, returning the result as a floating point value whether the input was floating or an integer.

stop *function 1 arg*

To quit Lisp you can call `stop` giving it an argument that is used as a return code from the system. This quits Lisp instantly and unconditionally and so should be used with some consideration.

stringp *function 1 arg*

Tests if its argument is a string. See `atom`, `numberp` and `symbolp`.

sub1 *function 1 arg*

Subtracts one from its argument.

subr *symbol*

See `expr`.

subst *function 3 args*

If you have a list or set of nested lists then `(subst a b c)` replaces every item in `c` that is equal to `b` with an `a`. In other words "substitute a for b in c". This only scans its input down to the level of atoms, so for instance vectors and hash tables do not have their components or contents looked at.

symbolp *function 1 arg*

Test if an onject is a symbol.

t *predefined variable*

In Lisp the symbol `nil` is used for *false*. If there is no better non-`nil` value to be used for *true* then `t` is used, and the symbol `t` starts off as a variable that has itself as its value.

tab *predefined variable*

A symbol whose initial value is a tab character. See `blank`.

terpri *function 0 args*

This TERminates the PRInt Line. It is equivalent to `(princ !`*eol*`!)`.

time *function 0 args*

If you wish to measure the amount of CPU time that some calculation takes then you can use `(time)` to read a timer both before and after. The difference between the two values will be the processor time used, measured in milliseconds.

times *special form*

Multiplies all of its arguments together.

trace *function 1 arg*

If you go `(trace '(f1 f2 ...))` then each of the functions `f1`,... is marked for tracing. When this has happened `vsl` prints messages each time the function concerned is called and each time it returns a result. This can be a great help when your code is misbehaving: you `trace` a suitable set of key functions and try some test examples. The extra output may be bulky but with luck will allow you to understand exactly what is happening. When fimished you can call `(untrace '(f1 f2 ...))` to restore things to their normal state.

upbv *function 1 arg*

Given a vector, `upbv` returns the highest legal subscript that can be used with it. See `mkvect`, `putv` and `getv`.

untrace *function 1 arg*

See `trace`.

vectorp *function 1 arg*

Tests if an object is a vector. See `mkvect`.

vsl *symbol in lispsystem!**

A predefined variable `lispsystem!*` has items in it that can give information about the Lisp system that is in use. Here the only information provided is the symbol `vsl` which (obviously) identifies the Lisp version.

while *macro*

`while` is a macro that is provided with a predicate and then a sequence of expressions to be evaluated repeatedly for so long as the predicate yields something non-`nil`.

```
(let ((n 1))
   (while (lessp n 1000000)
      (printc (list n "is too small"))
      (setq n (times 3 n)))
   n)
```

would return the first power of three that is at least a million, prionting reports on its progress.

wrs *function 1 arg*

If you need to direct output to somewhere other than the terminal (for instance to a file or pipe) then you can use `wrs` to select the relevant stream as the one to print to. `wrs` returns the previously selected stream, and often you will wish to save that so you can restore it later. See `open`, `rds` and `close`.

`zerop` *function 1 arg*

Tests to see if its argument is 0 or 0.0. See `onep`.

Index