CSL reference

A C Norman

November 18, 2011

1 Introduction

This is reference material for CSL. The Lisp identifiers mentioned here are the ones that are initially present in a raw CSL image. Some proportion of them are not really intended to be used by end-users but are merely the internal components of some feature.

2 Command-line options

The items shown here are the ones that are recognized on the CSL command line. In general an option that requires an argument can be written as either `-x yyy` or as `-xyyy`. Arguments should be case insensitive.

`--` If the application is run in console mode then its standard output could be redirected to a file using shell facilities. But the `--` directive (followed by a file name) redirects output within the Lisp rather than outside it. If this is done a very limited capability for sending progress or status reports to stderr (or the title-bar when running in windowed mode) remains via the `report!-right` function.

The `-w` option may frequently make sense in such cases, but if that is not used and the system tries to run in a window it will create it starting off minimised.

`--help` It is probably obvious what this option does! Note that on Windows the application was linked as a windows binary so it carefully creates a console to display the help text in, and organizes a delay to give people a chance to read it.

`--my-path` At some time I had felt the need for this option, but I now forget what I expected to use it for! It leads the executable to display the fully rooted name of the directory it was in and then terminate. It may be useful in some script?

`--texmacs` If CSL/Reduce is launched from texmacs this command-line flag should be used to arrange that the `texmacs` flag is set in `lispsystem!*`, and the code may then do special things.

`-a -a` is a curious option, not intended for general or casual use. If given it causes the `(batchp)` function to return the opposite result from normal! Without "attfamily -a" `(batchp)` returns `T` either if at least one file was specified on the command line, or if the standard input is "not a tty" (under some operating systems this makes sense – for instance the standard input might not be a "tty" if it is provided via file redirection). Otherwise (ie primary input is directly from a keyboard) `(batchp)` returns `nil`. Sometimes this judgement about how "batch" the current run is will be wrong or unhelpful, so `-a` allows the user to coax the system into better behaviour. I hope that this is never used!

`-b -b` tells the system to avoid any attempt to recolour prompts and input text. It will mainly be needed on X terminals that have been set up so that they use colours that make the defaults here unhelpful. Specifically white-on-black and so on. `-b` can be followed by colour specifications to make things yet more specific. It is supposed to be the idea that three colours can be specified after it for output, input and prompts, with the letters KRGYbMCW standing for blacK, Red, Green, Yellow, blue, Magenta, Cyan and White. This may not fully work yet!

`-c` Displays a notice relating to the authorship of CSL. Note that this is an authorship statement not a Copyright notice, because if any (L)GPL code is involved that would place requirements on what was displayed in a Copyright Notice.

`-d` A command line entry `-Dname=value` or `-D name=value` sets the value of the named lisp variable to the value (as a string). Note that the value set is a *string* so if you wish to retrieve it and use it as a symbold or number within your code you will have to perform some conversion.

`-e` A "spare" option used from time to time to activate experiments within CSL.

`-f` At one stage CSL could run as a socket server, and `-f portnumber` activated that mode. `-f-` used a default port, 1206 (a number inspired by an account number on Titan that I used in the 1960s). The code that supports this may be a useful foundation to others who want to make a network service out of this code-base, but is currently disabled.

`-g` In line with the implication of this option for C compilers, this enables a debugging mode. It sets a lisp variable `!*backtrace` and arranges that all backtraces are displayed notwithstanding use of `errorset`.

`-h` This option is a left-over. When the X-windows version of the code first started to use Xft it viewed that as optional and could allow a build even when it was not available. And then even if Xft was detected and liable to be used by default it provided this option to disable its use. The remnants of the switch that disabled use of Xft (relating to fonts living on the Host or the Server) used this switch, but it now has no effect.

`-i` CSL and Reduce use image files to keep both initial heap images and "fasl" loadable modules. By default if the executable launched has some name, say xxx, then an image file xxx.img is used. But to support greater generality `-i` introduces a new image, `-i-` indicates the default one and a sequence of such directives list image files that are searched in the order given. These are read-only. The similar option `-o` equally introduces image files that are scanned for input, but that can also be used for output. Normally there would only be one `-o` directive.

`-j` Follow this directive with a file-name, and a record of all the files read during the Lisp run will be dumped there with a view that it can be included in a Makefile to document dependencies.

`-k -K nnn` sets the size of heap to be used. If it is given then that much memory will be allocated and the heap will never expand. Without this option a default amount is used, and (on many machines) it will grow if space seems tight.

The extended version of this option is `-K nnn/ss` and then ss is the number of "CSL pages" to be allocated to the Lisp stack. The default value (which is 1) should suffice for almost all users, and it should be noted that the C stack is separate from and independent of this one and it too could overflow.

A suffix K, M or G on the number indicates units of kilobytes, megabytes or gigabytes, with megabytes being the default. So `-K200M` might represent typical usage for common-sized computations. In general CSL will automatically expand its heap, and so it should normally never be necessary to use this option.

`-l` This is to send a copy of the standard output to a named log file. It is very much as if the Lisp function `(spool ''logfile'')` had been invoked at the start of the run.

`-m` Memory trace mode. An option that represents an experiment from the past, and no longer reliably in use. It make it possible to force an exception at stages whene reference to a specified part of memory was made and that could be useful for some low level debugging. It is not supported at present.

`-n` Normally when the system is started it will run a "restart function" as indicated in its heap image. There can be cases where a heap image has been created in a bad way such that the saved restart function always fails abruptly, and hence working out what was wrong becomes hard. In such cases it may be useful to give the `-n` option that forces CSL to ignore any startup function and merely always begin in a minimal Lisp-style read-eval-print loop. This is intended for experts to do disaster recovery and diagnosis of damaged image files.

`-o` See `-i`. This specifies an image file used for output via `faslout` and `reserve`.

`-p` If a suitable profile option gets implemented one day this will activate it, but for now it has no effect.

`-q` This option sets `!*echo` to `nil` and switches off garbage collector messages to give a slightly quieter run.

`-r` The random-number generator in CSL is normally initialised to a value based on the time of day and is hence not reproducible from run to run. In many cases that behavious is desirable, but for debugging it can be useful to force a seed. The directive `-r nnn,mmm` sets the seed to up to 64 bits taken from the values nnn and mmm. The second value if optional, and specifying `-r0` explicitly asks for the non-reproducible behaviour (I hope). Note that the main Reduce-level random number source is coded at a higher level and does not get reset this way – this is the lower level CSL generator.

`-s` Sets the Lisp variable `!*plap` and hence the compiler generates an assembly listing.

`-t -t name` reports the time-stamp on the named module, and then exits. This is for use in perl scripts and the like, and is needed because the stamps on modules within an image or library file are not otherwise instantly available.

Note that especially on windowed systems it may be necessary to use this with `-- filename` since the information generated here goes to the default output, which in some cases is just the screen.

`-u` See `-d`, but this forcibly undefines a symbol. There are probably very very few cases where it is useful since I do not have a large number of system-specific predefined names.

`-v` An option to make things mildly more verbose. It displays more of a banner at startup and switches garbage collection messages on.

`-w` On a typical system if the system is launched it creates a new window and uses its own windowed intarface in that. If it is run such that at startup the standard input or output are associated with a file or pipe, or under X the variable `DISPLAY` is not set it will try to start up in console mode. The flag `-w` indicates that the system should run in console more regadless, while `-w+` attempts a window even if that seems doomed to failure. When running the system to obey a script it will often make sense to use the `-w` option. Note that on Windows the system is provided as two separate (but almost identical) binaries. For example the file `csl.exe` is linked in windows mode. A result is that if launched from the command line it detaches from its console, and if launched by double-clicking it does not create a console. It is in fact very ugly when double clicking on an application causes an unwanted console window to appear. In contrast `csl.com` is a console mode version of just the same program, so when launched from a command line it can communicate with the console in the ordinary expected manner.

`-x -x` is an option intended for use only by system support experts – it disables trapping if segment violations by errorset and so makes it easier to track down low level disasters – maybe! This can be valuable when running under a debugger since if the code traps signals in its usual way and tries to recover it can make it a lot harder to find out just what was going wrong.

`-y -y` sets the variable `!*hankaku`, which causes the lisp reader convert a Zenkaku code to Hankaku one when read. I leave this option decoded on the command line even if the Kanji support code is not otherwise compiled into CSL just so I can reduce conditional compilation. This was part of the Internationalisation effort for CSL bu this is no longer supported.

`-z` When bootstrapping it is necessary to start up the system for one initial time without the benefit of any image file at all. The option `-z` makes this happen, so when it is specified the system starts up with a minimal environment and only those capabilities that are present in the CSL kernel. It will normally make sense to start loading some basic Lisp definitions rather rapidly. The files `compat.lsp`, `extras.lsp` and `compiler.lsp` have Lisp source for the main things I use, and once they are loaded the Lisp compiler can be used to compile itself.

3 Predefined variables

`!!fleps1` There is a function safe!-fp!-plus that performs floating point arithmetic but guarantees never to raise an exception. This value

was at one stage related to when small values created there got truncated to zero, but the current code does not use the Lisp variable at all and instead does things based on the bitwise representation of the numbers.

`!$eof!$` The value of this variable is a pseudo-character returned from various read functions to signal end-of-file.

`!$eol!$` The value of this variable is an end-of-line character.

`!*plap` Not yet written

`!*applyhook!*` If this is set it might be supposed to be the name of a function used by the interpreter as a callbackm but at presnet it does not actually do anything!

`!*break!-loop!*` If the value of this is a symbol that is defined as a function of one argument then it is called during the processing on an error. This has not been used in anger and so its whole status may be dubious!

`!*carcheckflag` In general CSL arranges that every `car` or `cdr` access is checked for validity. Once upon a time setting this variable to nil turned such checks off in the hope of gaining a little speed. But it no longer does that. It may have a minor effect on array access primitives.

`!*comp` When set each function is compiled (into bytecodes) as it gets defined.

`!*debug!-io!*` An I/O channel intended to be used for diagnostic interactions.

`!*echo` When this is non-nil characters that are read from an input file are echoed to the standard output. This gives a more comlete transcript in a log file, but can sometimes amount to over-verbose output.

`!*error!-messages!*` Has the value nil and does not do anything!

`!*error!-output!*` An I/O channel intended for diagnostic output.

`!*evalhook!*` See `!*applyhook!*`. This also does not do anything at present.

`!*gc!-hook!*` If this is set to have as its value that is a function of one argument then that function is called with `nil` on every minor entry to the garbage collection, and with argument `t` at the end of a "genuine" full garbage collection.

`!*hankaku` This was concerned with internationalisation to support a Japanese locale but has not been activated for some while. In the fullness of time I hope to migrate CSL to use an UTF8 representation of Unicode characters internally, but that upgrade is at present an ideal and a project not a reality. Volunteers to help welcomed.

`!*loop!-print!*` Probably not used at present.

`!*lower` Not yet written

`!*macroexpand!-hook!*` Not yet written

`!*math!-output!*` Not yet written

`!*native_code` Not yet written

`!*notailcall` Not yet written

`!*package!*` Not yet written

`!*pgwd` Not yet written

`!*pretty!-symmetric` Not yet written

`!*prinl!-fn!*` Not yet written

`!*prinl!-index!*` Not yet written

`!*prinl!-visited!-nodes!*` Not yet written

`!*print!-array!*` Not yet written

`!*print!-length!*` Not yet written

`!*print!-level!*` Not yet written

`!*pwrds` Not yet written

`!*query!-io!*` Not yet written

`!*quotes` Not yet written

`!*raise` Not yet written

`!*redefmsg` Not yet written

`!*resources!*` Not yet written

`!*savedef` Not yet written

`!*spool!-output!*` Not yet written

`!*standard!-input!*` Not yet written

`!*standard!-output!*` Not yet written

`!*terminal!-io!*` Not yet written

`!*trace!-output!*` Not yet written

`!@cslbase` Not yet written

]pendingrpars]

`pendingrpars` Not yet written

`blank` The value of this variable is an space or blank character. This might otherwise be written as "`!  `".

`bn` Not yet written

`bufferi` Not yet written

`bufferp` Not yet written

`common!-lisp!-mode` Not yet written

`crbuf!*` Not yet written

`emsg!*` Not yet written

`eof!*` Not yet written

`esc!*` The value of this variable is the character "escape". As a non-printing character use of this is to be viewed as delicate.

`indblanks` Not yet written

`indentlevel` Not yet written

`initialblanks` Not yet written

`lispsystem!*` Not yet written

`lmar` Not yet written

`load!-source` Not yet written

`nil` Not yet written

`ofl!*` Not yet written

`program!*` Not yet written

`rmar` Not yet written

`rparcount` Not yet written

`s!:gensym!-serial` Not yet written

`stack` Not yet written

`t` Not yet written

`tab` The value of this variable is a tab character.

`thin!*` Not yet written

`ttype!*` Not yet written

/*!! flags [04] Flags and Properties

Most of tags here are probably not much use to end-users, but I am noting them as a matter of completeness.

Items that can appear in `lispsystem!*`

There is a global variable called `lispsystem!*` whose value is reset in the process of CSL starting up. An effect of this is that if the user changes its value those changes do not survice a preserving and re-loading a heap image: this is deliberate since the heap image may be re-loaded on a different instance of CSL possibly on a quite different computer of with a different configuration. The value of `lispsystem!*` is a list of items, where each item is either an atomic tag of a pair whose first component is a key. In general it would be unwise to rely on exactly what information is present without review of the code that sets it up. The information may be of interest to anybody but some tags and keys are reflections of experiments rather than fully stable facilities.

`(c!-code . count)` This will be present if code has been optimised into C through the source files u01.c to u60.c, and in that case the value tells you how many functions have been optimised in this manner.

`common!-lisp` For a project some while ago a limited Common Lisp compatibility mode was being developed, and this tag indicated that it was active. In that case all entries are in upper case and the variable is called `*FEATURES*` rather than `lispsystem!*`. But note that this Lisp has never even aspired to be a full Common Lisp, since its author considers Common Lisp to have been a sad mistake that must bear significant responsibility for the fact that interest in Lisp has faded dramatically since its introduction.

`(compiler!-command . command)` The value associated with this key is a string that was used to compile the files of C code making up CSL. It should contain directives to set up search paths and predefined symbols. It is intended to be used in an experiment that generates C code

synamically, uses a command based on this string to compile it and then dynamically links the resulting code in with the running system.

`csl` A simple tag intended to indicate that this Lisp system is CSL and not any other. This can of course only work properly if all other Lisp systems agree not to set this tag! In the context of Reduce I note that the PSL Lisp system sets a tag `psl` on `lispsystem!*` and the realistic use of this is to discriminate between CSL and PSL hosted copies of Reduce.

`debug` If CSL was compiled with debugging options this is present, and one can imagine various bits of code being more cautious or more verbose if it is detected.

`(executable . name)` The value is the fully rooted name of the executable file that was launched.

`fox` Used to be present if the FOX GUI toolkit was detected and incorporated as part of CSL, but now probably never used!

`(linker . type)` Intended for use in association with `compiler!-command`, the value is `win32` on Windows, `x86_64` on 64-bit Linux and other things on other systems, as detected using the program `objtype.c`.

`(name . name)` Some indication of the platform. For instance on one system I use it is `linux-gnu:x86_64` and on anther it is just `win32`.

`(native . tag)` One of the many experiments within CSL that were active at one stage but are not current involved compilation directly into machine code. The strong desire to ensure that image files coudl be used on a cross-platform basis led to saved compiled code being tagged with a numeric "native code tag", and this key/value pair identified the value to be used on the current machine.

`(opsys . operating-system)` Some crude indication of the host operating system.

operating system identity The name of the current operating system is put on the list. Exactly what form is not explicitly defined!

`pipes` In the earlier days of CSL there were computers where pipes were not supported, so this tag notes when they are present and hance the facility to create sub-tasks through them can be used.

`record_get` An an extension to the CSL profiling scheme it it possible to compile a special version that tracks and counts each use of property-list access functions. This can be useful because there are ways to give special treatment to a small number of flags and a small number of

properties. The special-case flage end up stored as a bitmap in the symbol-header so avoid need for property-list searching. But of course recording this extra information slows things down. This tag notes when the slow version is in use. It might be used to trigger a display of statistics at the end of a calculation.

`reduce` This is intended to report if the initial heap image is for Reduce rather than merely for Lisp.

`(shortname . name)` Gives the short name of the current executable, without its full path.

`showmath` If the "showmath" capability has been compiled into CSL this will be present so that Lisp code can know it is reasonable to try to use it.

`sixty!-four` Present if the Lisp was compiled for a 64-bit computer.

`termed` Present if a cursor-addressable console was detected.

`texmacs` Present if the system was launched with the `--texmacs` flag. The intent is that this should only be done when it has been launched with texmacs as a front-end.

`(version . ver)` The CSL version number.

`win32`, `win64` Any windows system puts `win32` in `lispsystem!*`. If 64-bit windows is is use then `win64` is also included

`windowed` Present if CSL is running in its own window rather than in console mode.

4 Flags and Properties

`lose` If a name is flagged as ttfamily lose then a subsequent attempt to define or redefine it will be ignored.

`s!:ppchar` **and** `s!:ppformat` These are used in the prettyprint code found in `extras.red`. A name is given a property `s!:ppformat` if in prettyprinted display its first few arguments should appear on the same line as it if at all possible. The `s!:ppchar` property is used to make the display of bracket characters a little more tide in the source code.

`switch` In the Reduce parser some names are "switches", and then directives such as `on xxx` and `off xx` have the effect of setting or clearing the value of a variable `!*xxx`. This is managed by setting the `switch` flag om `xxx`. CSL sets some things as switches ready for when they may be used by the Reduce parser.

`!~magic!-internal!-symbol!~` CSL does not have a clear representation for functions that is separated from the representation of an identifier, and so when you ask to get the value of a raw function you get an identifier (probably a gensym) and this tag is used to link such values with the symbols they were originally extracted from.

5 Functions and Special Forms

Each line here shows a name and then one of the words *expr*, *fexpr* or *macro*. In some cases there can also be special treatment of functions by the compiler so that they get compiled in-line.

`abs` ***expr***
Not yet written

`binary_close_input` ***expr***
Not yet written

`binary_close_output` ***expr***
Not yet written

`binary_open_input` ***expr***
Not yet written

`binary_open_output` ***expr***
Not yet written

`binary_prin1` ***expr***
Not yet written

`binary_prin2` ***expr***
Not yet written

`binary_prin3` ***expr***
Not yet written

`binary_prinbyte` ***expr***
Not yet written

`binary_princ` ***expr***
Not yet written

`binary_prinfloat` ***expr***
Not yet written

`binary_read2` ***expr***
Not yet written

`binary_read3` ***expr***
: Not yet written

`binary_read4` ***expr***
: Not yet written

`binary_readbyte` ***expr***
: Not yet written

`binary_readfloat` ***expr***
: Not yet written

`binary_select_input` ***expr***
: Not yet written

`binary_terpri` ***expr***
: Not yet written

`bps!-getv` ***expr***
: Not yet written

`bps!-putv` ***expr***
: Not yet written

`bps!-upbv` ***expr***
: Not yet written

`break!-loop` ***expr***
: Not yet written

`c_out` ***expr***
: Not yet written

`caaaar` ***expr***
: see `caar`.

`caaadr` ***expr***
: see `caar`.

`caaar` ***expr***
: see `caar`.

`caadar` ***expr***
: see `caar`.

`caaddr` ***expr***
: see `caar`.

`caadr` ***expr***
: see `caar`.

`caar ...cddddr` ***expr***

Names that start with `c`, then have a sequence of `a` or `d`s and finally `r` provide shorthand functions for chains of uses of `car` and `cdr`. Thus for instance `(cadar x)` has the same meaning as `(car (cdr (car x)))`.

`cadaar` ***expr***

see `caar`.

`cadadr` ***expr***

see `caar`.

`cadar` ***expr***

see `caar`.

`caddar` ***expr***

see `caar`.

`cadddr` ***expr***

see `caar` and `fourth`.

`caddr` ***expr***

see `caar` and `third`.

`cadr` ***expr***

see `caar` and `second`.

`car` ***expr***

For a non-empty list the function `car` will return the first element. For a dotted pair (created using `cons`) it extracts the first component. This is the fundamental low-level data structure access function in Lisp. See `cdr` for the function that returns the tail or a list or the second component of a dotted pair. In CSL any attempt to tape `car` of an atom should be detected and will be treated as an error. If CSL had been compiled in Common Lisp mode (which is now not probable) a special exemption would apply and `car` and cdr of the empty lisp `nil` would be `nil`.

`car!*` ***expr***

This function behaves like `car` except that if its argument is atomic then the argument is returned unaltered rather than that case being treated as an error.

`cdaaar` ***expr***

see `caar`.

`cdaadr` ***expr***

see `caar`.

`cdaar` ***expr***
see `caar`.

`cdadar` ***expr***
see `caar`.

`cdaddr` ***expr***
see `caar`.

`cdadr` ***expr***
see `caar`.

expr
see `caar`.

`cddaar` ***expr***
see `caar`.

`cddadr` ***expr***
see `caar`.

`cddar` ***expr***
see `caar`.

`cdddar` ***expr***
see `caar`.

`cddddr` ***expr***
see `caar`.

`cdddr` ***expr***
see `caar`.

`cddr` ***expr***
see `caar`.

`cdr` ***expr***
See `car`.

`char!-code` ***expr***
Not yet written

`char!-downcase` ***expr***
Not yet written

`char!-upcase` ***expr***
Not yet written

`check!-c!-code` ***expr***
Not yet written

`cl!-equal` ***expr***
Not yet written

`close!-library` ***expr***
Not yet written

`code!-char` ***expr***
Not yet written

`compile!-all` ***expr***
Not yet written

`convert!-to!-evector` ***expr***
Not yet written

`copy!-module` ***expr***
Not yet written

`copy!-native` ***expr***
Not yet written

`create!-directory` ***expr***
Not yet written

`dated!-name` ***expr***
Not yet written

`define!-in!-module` ***expr***
Not yet written

`delete!-file` ***expr***
Not yet written

`delete!-module` ***expr***
Not yet written

`do!*` ***macro***
Not yet written

`double!-execute` ***expr***
Not yet written

`enable!-backtrace` ***expr***
Not yet written

`enable!-errorset` ***expr***
Not yet written

`eq!-safe` ***expr***
Not yet written

`eval!-when` ***fexpr***
Not yet written

`file!-length` ***expr***
Not yet written

`file!-readablep` ***expr***
Not yet written

`file!-writeablep` ***expr***
Not yet written

`flagp!*!*` ***expr***
Not yet written

`fp!-evaluate` ***expr***
Not yet written

`funcall!*` ***expr***
Not yet written

`get!*` ***expr***
Not yet written

`get!-current!-directory` ***expr***
Not yet written

`get!-lisp!-directory` ***expr***
Not yet written

`hash!-table!-p` ***expr***
Not yet written

`hashtagged!-name` ***expr***
Not yet written

`input!-libraries` ***fexpr***
Not yet written

`instate!-c!-code` ***expr***
Not yet written

`internal!-open` ***expr***
Not yet written

`is!-console` ***expr***
Not yet written

`let!*` ***fexpr***
Not yet written

`library!-members` ***expr***

Returns a list of all the modules that could potentially be loaded using `load!-module`. See `list!-modules` to get a human readable display that looks more like the result of listing a directory, or `modulep` for checking the state of a particular named module.

`library!-name` ***expr***

Not yet written

`list!*` ***fexpr***

Not yet written

`list!-directory` ***expr***

Not yet written

`list!-modules` ***expr***

This prints a human-readable display of the modules present in the current image files. This will include "InitialImage" which is the heap-image loaded at system startup. For example

```
> (list!-modules)

File d:\csl\csl.img (dirsize 8  length 155016, Writable):
  compat       Sat Jul 26 10:20:08 2008  position 556   size: 9320
  compiler     Sat Jul 26 10:20:08 2008  position 9880  size: 81088
  InitialImage Sat Jul 26 10:20:09 2008  position 90972 size: 64040

nil
```

See `library!-members` and `modulep` for functions that make it possible for Lisp code to discover about the loadable modules that are available.

`list!-to!-string` ***expr***

Not yet written

`list!-to!-symbol` ***expr***

Not yet written

`list!-to!-vector` ***expr***

Not yet written

`list2!*` ***expr***

Not yet written

`list3!*` ***expr***

Not yet written

`load!-module` ***expr***
: Not yet written

`load!-source` ***expr***
: Not yet written

`lose!-precision` ***expr***
: Not yet written

`macro!-function` ***expr***
: Not yet written

`macroexpand!-1` ***expr***
: Not yet written

`make!-bps` ***expr***
: Not yet written

`make!-function!-stream` ***expr***
: Not yet written

`make!-global` ***expr***
: Not yet written

`make!-native` ***expr***
: Not yet written

`make!-random!-state` ***expr***
: Not yet written

`make!-simple!-string` ***expr***
: Not yet written

`make!-special` ***expr***
: Not yet written

`math!-display` ***expr***
: Not yet written

`member!*!*` ***expr***
: Not yet written

`rplaca` ***expr***
: This is a destructive function in that it alters the data structure that it is given as its first argument by updating its `car` component. The result is the updated object. See `rplacd` for the corresponding function for updating the `cdr` component.

`rplacd` ***expr***
: See `rplaca`

`!~block` ***fexpr***
: Not yet written

`!~let` ***fexpr***
: Not yet written

`!~tyi` ***expr***
: Not yet written