Ralph's Common Lisp Library

Edition 1 (draft), 2013-10-08, for RS-CLL version 20131008.0250

Ralph Schleicher

This is edition 1 (draft), last updated 2013-10-08, of *Ralph's Common Lisp Library Reference Manual*, for RS-CLL version 20131008.0250.

Please report any errors in this manual to `rs@ralph-schleicher.de`.

Table of Contents

1 Introduction

Ralph's Common Lisp Library, RS-CLL for short, is a collection of routines for programming in Common Lisp.

2 Basic Definitions

2.1 Data

`defconst` *name value* &optional *doc* [Macro]
Define a constant variable.

This is like `defconstant` except that the initially set value is reused when the `defconst` form is evaluated again.

`defsubst` *name arg-list* &body *body* [Macro]
Define an inline function.

This is like `defun` except that the function is globally marked for inline expansion by the compiler.

`false` &rest *arguments* [Function]
Ignore all arguments and return nil.

`true` &rest *arguments* [Function]
Ignore all arguments and return t.

`nothing` &rest *arguments* [Function]
Ignore all arguments and return no values.

2.2 Types

`list-of-strings-p` *object* [Function]
Return true if *object* is a list of strings.

If *object* is the empty list, value is true, too.

`list-of-strings` [Type]
Type specifier for a list of strings.

2.3 Conditions

`ensure-type` *object type* [Function]
Signal a `type-error` if *object* is not of the type *type*. Otherwise, return *object*.

2.4 Symbols

`symbol-name*` *symbol* [Function]
Return the name of *symbol* including the package prefix.

If *symbol* does not belong to a package, just return it's name. If *symbol* is a keyword, add a leading colon character to the name. If *symbol* is an external symbol of a package, seperate the package name and the symbol name by a colon character. Otherwise, seperate the package name and the symbol name by two colon characters.

2.5 Numbers

2.5.1 Numerical Constants

`pi/6` [Constant]
One sixth of the ratio of a circle's circumference to its diameter. This is equal to 30 arc degree.

`pi/4` [Constant]
One quarter of the ratio of a circle's circumference to its diameter. This is equal to 45 arc degree.

`pi/2` [Constant]
One half of the ratio of a circle's circumference to its diameter. This is equal to 90 arc degree.

`2*pi` [Constant]
Two times the ratio of a circle's circumference to its diameter. This is equal to 360 arc degree.

2.5.2 Mathematical Operators

Minimum and Maximum

`minf` *place* &rest *numbers* [Macro]
Set the value designated by *place* to the minimum of the current value and *numbers*.

`maxf` *place* &rest *numbers* [Macro]
Set the value designated by *place* to the maximum of the current value and *numbers*.

Basic Arithmetics

`addf` *place* &rest *numbers* [Macro]
Add *numbers* to the value designated by *place*.

`subf` *place* &rest *numbers* [Macro]
Subtract *numbers* to the value designated by *place*. If *numbers* is omitted, change the sign of the value.

`mulf` *place* &rest *numbers* [Macro]
Multiply the value designated by *place* by *numbers*.

`divf` *place* &rest *numbers* [Macro]
Divide the value designated by *place* by *numbers*. If *numbers* is omitted, invert the value.

Combined Arithmetics

`fma` *number multiplicand summand* [Function]
Multiply *number* by *multiplicand*, then add *summand*. Attempt to perform a fused multiply-add operation.

This is the inverse of the `fsd` function.

`fsd` *number subtrahend divisor* [Function]
Subtract *subtrahend* from *number*, then divide by *divisor*. Attempt to perform a fused subtract-divide operation.

This is the inverse of the `fma` function.

`fmaf` *place multiplicand summand* [Macro]
Multiply the value designated by *place* by *multiplicand*, then add *summand*. Attempt to perform a fused multiply-add operation.

`fsdf` *place subtrahend divisor* [Macro]
Subtract *subtrahend* from the value designated by *place*, then divide by *divisor*. Attempt to perform a fused subtract-divide operation.

2.5.3 Floating-Point Numbers

`float-epsilon` &optional *float* [Function]
Return the smallest positive floating-point number used in the representation of *float*, such that the expression

```
(= (float 1 epsilon) (+ (float 1 epsilon) epsilon))
```

is false.

`float-negative-epsilon` &optional *float* [Function]
Return the smallest positive floating-point number used in the representation of *float*, such that the expression

```
(= (float 1 epsilon) (- (float 1 epsilon) epsilon))
```

is false.

`float-digits*` *float* &optional *base* [Function]
Return the number of digits used in the representation of *float*. This includes any implicit digits.

Optional argument *base* is the radix for the return value. Default is 10.

If *base* is equal to the radix of *float*, value is an integral number. Otherwise, value is a floating-point number in the format of *float*.

`float-precision*` *float* &optional *base* [Function]
Return the number of significant digits present in *float*.

Optional argument *base* is the radix for the return value. Default is 10.

If *base* is equal to the radix of *float*, value is an integral number. Otherwise, value is a floating-point number in the format of *float*. If *float* is zero, value is zero.

`decimal-digits` *float* [Function]
Return the number of decimal digits needed to preserve the original floating-point number when converting it to a decimal character format.

2.5.4 Sorting Numbers

`absolute-ascending` *a b* [Function]
Return true if the absolute value of number *a* is less than the absolute value of number *b*. This function can be used to sort numbers in ascending order.

`absolute-descending` *a b* [Function]
Return true if the absolute value of number *a* is greater than the absolute value of number *b*. This function can be used to sort numbers in descending order.

2.5.5 Number Systems

`roman-numeral` *n* [Function]
Convert the integral number *n* into a roman number (a string). If *n* is zero, the return value is nil. If *n* is a negative number, utilize lowercase letters. Otherwise, use uppercase letters.

2.6 Quantities

`radian-from-degree` *deg* [Function]
Convert a plane angle from degree to radian.

- Argument *deg* is the angle given in degree.

Value is the corresponding angle given in radian.

`degree-from-radian` *rad* [Function]
Convert a plane angle from radian to degree.

- Argument *rad* is the angle given in radian.

Value is the corresponding angle given in degree.

`degree-from-sexagesimal` *deg* &optional *min s sign* [Function]
Join sexagesimal subdivisions of an arc degree into a plane angle.

- First argument *deg* is the number of arc degrees.
- Optional second argument *min* is the number of arc minutes.
- Optional third argument *s* is the number of arc seconds.
- Optional fourth argument *sign* specifies the sign.

Arguments *deg*, *min*, and *s* have to be non-negative numbers.

Value is the corresponding angle given in degree. If none of the arguments is a floating-point number, value is a rational number.

`sexagesimal-from-degree` *angle* [Function]
Split a plane angle into sexagesimal subdivisions of an arc degree.

- Argument *angle* is the angle given in degree.

Return values are the arc degrees, arc minutes, and arc seconds. The number of arc seconds may be a floating point number with fractions of a second. Fourth value is either plus one or minus one specifying the sign of the angle.

2.7 Characters

`whitespace-char-p` *char* [Function]
Return true if *char* is a whitespace character.

Argument *char* has to be a character object.

`blank-char-p` *char* [Function]
Return true if *char* is a space or horizontal tab character.

Argument *char* has to be a character object.

`standard-alpha-char-p` *char* [Function]
Return true if *char* is a standard alphabetic character.

Argument *char* has to be a character object.

`standard-digit-char-p` *char* [Function]
Return true if *char* is a standard digit character.

- First argument *char* has to be a character object.
- Optional second argument *radix* is an integer between 2 and 36, inclusive. Default is 10.

Value is the weight of *char* as an integer, or nil.

2.8 Arrays

`linear-index-from-subscripts` *dimensions subscripts* [Function]
Return the linear index corresponding to a set of subscript values.

- First argument *dimensions* is a list of valid array dimensions.
- Second argument *subscripts* is a list of valid array indices.

The return value of the `linear-index-from-subscripts` function is equal to the value of the form

```
(apply #'array-row-major-index (make-array dimensions) subscripts)
```

`subscripts-from-linear-index` *dimensions index* [Function]
Return the set of subscript values corresponding to a linear index.

- First argument *dimensions* is a list of valid array dimensions.
- Second argument *index* is a valid array index.

This is the inverse function of `linear-index-from-subscripts`.

2.9 Strings

`*random-string-alphabet*` [Parameter]
The character set for generating random strings. Value has to be a vector.

`random-string` *n* [Function]
Return a string with *n* random characters.

2.10 Sequences

`start-index-if` *predicate seq* &key *start end key* [Function]
Return start index of first element in *seq* matching *predicate*.

- Keywords *start* and *end* are bounding index designators.
- Keyword *key* is a function designator of one argument.

If no element matches *predicate*, return the end index position. Likewise if *seq* is empty.

`start-index-if-not` *predicate seq* &key *start end key* [Function]
Return start index of first element in *seq* not matching *predicate*.

- Keywords *start* and *end* are bounding index designators.
- Keyword *key* is a function designator of one argument.

If all elements match *predicate*, return the end index position. Likewise if *seq* is empty.

`end-index-if` *predicate seq* &key *start end key* [Function]
Return end index of last element in *seq* matching *predicate*.

- Keywords *start* and *end* are bounding index designators.
- Keyword *key* is a function designator of one argument.

If no element matches *predicate*, return the start index position. Likewise if *seq* is empty.

`end-index-if-not` *predicate seq* &key *start end key* [Function]
Return end index of last element in *seq* not matching *predicate*.

- Keywords *start* and *end* are bounding index designators.
- Keyword *key* is a function designator of one argument.

If all elements match *predicate*, return the start index position. Likewise if *seq* is empty.

`bounding-indices-if` *predicate seq* &key *start end key* [Function]
Return start index of first element in *seq* and end index of last element in *seq* matching *predicate*.

- Keywords *start* and *end* are bounding index designators.
- Keyword *key* is a function designator of one argument.

If no element matches *predicate*, return the end index positions. Likewise if *seq* is empty.

`bounding-indices-if-not` *predicate seq* &key *start end key* [Function]
Return start index of first element in *seq* and end index of last element in *seq* not matching *predicate*.

- Keywords *start* and *end* are bounding index designators.
- Keyword *key* is a function designator of one argument.

If all elements match *predicate*, return the end index positions. Likewise if *seq* is empty.

2.11 Streams

`read-file` &optional *input-stream element-length* [Function]
Read all elements from a stream.

- Optional first argument *input-stream* has to be an open input stream. The default is standard input.
- Optional second argument *element-length* is the size of an element in the external format given in byte. This argument has no effect for binary streams.

If the element type of the stream is a character, value is a string. Otherwise, value is a vector.

The *element-length* argument may be used to speed up reading from the stream by allocating the resulting sequence all at once. This only works if the file size of the stream can be determined, too. If the external format has a variable length element size like, for example, the UTF-8 character encoding, then *element-length* should be the minimum length of an element. In such a case, more memory than actually needed may be allocated for the return value.

Reading a text file:

```
(with-open-file (s ... :direction :input)
  (read-file s))

(with-open-file (s ... :direction :input :external-format :UTF-8)
  (read-file s 1))

(with-open-file (s ... :direction :input :external-format :UCS-2BE)
  (read-file s 2))
```

Reading a binary file:

```
(with-open-file (s ... :direction :input :element-type '(unsigned-byte 8))
  (read-file s))"
```

2.12 Reader

`q-reader` *stream sub-char arg* [Function]
Read a quoted text.

First character is the delimiting character. Non-bracketing delimiters use the same character fore and aft. Round brackets, angle brackets, square brackets, and curly brackets always match the other character of the pair.

Without prefix argument, bracketing delimiters nest. If prefix argument is zero, bracketing delimiters use the same character fore and aft, that means they are treated like a non-bracketing delimiting character. If prefix argument is a postive number, bracketing delimiters do not nest, that means the first matching other character of the pair terminates reading.

You can enable this feature by evaluating the form

```
(set-dispatch-macro-character #\# #\q #'q-reader)
```

After that, you can read literal strings via

```
#q|He said "Quote me!"|
```

or

```
#q<<tag attr="val">text</tag>>
```

but

```
#q(")")
```

will fail, because the `q-reader` function is not a parser for any particular grammar.

3 Mathematics

3.1 Trigonometric Functions

`hypot` *x y* [Function]
Return the distance between a point and the origin in a two-dimensional Cartesian coordinate system.

Arguments *x* and *y* have to be real numbers.

`hypot3` *x y z* [Function]
Return the distance between a point and the origin in a three-dimensional Cartesian coordinate system.

Arguments *x*, *y*, and *z* have to be real numbers.

3.2 Exponential Functions

`cbrt` *number* [Function]
Return the cube root of *number*.

If *number* is a real number, value is the real cube root of *number*.

`square` *number* [Function]
Return *number* squared, that is *number* raised to the power two.

Argument *number* has to be a number.

`square-root` *number* [Function]
Return the square root of *number*.

Argument *number* has to be a number.

The `square-root` function attempts to propagate the type of the argument *number* to its value.

`cube` *number* [Function]
Return *number* cubed, that is *number* raised to the power three.

Argument *number* has to be a number.

`cube-root` *number* [Function]
Return the cube root of *number*.

Argument *number* has to be a number.

If argument *number* is zero, value is zero. If argument *number* is a real number, value is the real cube root of *number*.

The `cube-root` function attempts to propagate the type of the argument *number* to its value.

3.3 Optimization

`brent` *f* &key *y initial-value lower-bound upper-bound max-iter rel-tol abs-tol* [Function]
Solve univariate function $y = f(x)$ using Brent's method.

- First argument *f* is a function taking one numeric argument. Return value of *f* has to be a real number.
- Keyword argument *y* is the function value. Default is zero.
- Keyword argument *initial-value* is the initial value for the function argument *x*.

- Keyword arguments *lower-bound* and *upper-bound* specify the interval bounds between which the root is searched. If any one of these two arguments is omitted, the interval bounds are determined automatically around *initial-value*.
- Keyword argument *max-iter* is the maximum number of iterations to be performed. Default is 1000.
- Keyword argument *rel-tol* is the relative tolerance. Default is machine precision.
- Keyword argument *abs-tol* is the absolute tolerance. Default is zero.

The iteration stops if half of the interval is less than or equal to $2\mathit{rel\text{-}tol}|x| + \mathit{abs\text{-}tol}/2$.

Primary value is the argument x for which $y = f(x)$ is true. Secondary value is the number of iterations performed; nil means that the algorithm did not converge within the given number of iterations.

3.4 Polynomials

`evaluate-polynomial` *coefficients number* [Function]

Evaluate a polynomial.

- First argument *coefficients* is a sequence whose elements are the coefficients of the polynomial in descending order.
- Second argument *number* is the argument at which the polynomial is evaluated.

`quadratic-formula-1` *p q* [Function]

Calculate the roots of a monic quadratic function

$$f(x) = x^2 + p\,x + q$$

Arguments p and q are the coefficients of the polynomial.

Value is a list of two numbers.

`quadratic-formula` *a b c* [Function]

Calculate the roots of a general quadratic function

$$f(x) = a\,x^2 + b\,x + c$$

Arguments a, b, and c are the coefficients of the polynomial.

Value is a list of two numbers.

`cubic-formula-1` *p q r* [Function]

Calculate the roots of a monic cubic function

$$f(x) = x^3 + p\,x^2 + q\,x + r$$

Arguments p, q, and r are the coefficients of the polynomial.

Value is a list of three numbers. The first element is a real number.

`cubic-formula` *a b c d* [Function]

Calculate the roots of a general cubic function

$$f(x) = a\,x^3 + b\,x^2 + c\,x + d$$

Arguments a, b, c, and d are the coefficients of the polynomial.

Value is a list of three numbers.

4 Miscellaneous

4.1 Regular Expressions

The symbols documented in this section are an extension to the CL-PPCRE package. It provides caching of compiled regular expressions and some syntactic sugar for working with the match data.

`string-match` *regexp string* &key *start end* [Function]
Return start position of first match for *regexp* in *string*, or nil if there is no match.

- First argument *regexp* is a regular expression.
- Second argument *string* is the target string.
- Keyword arguments *start* and *end* are bounding indices in *string*. Default values are zero and the length of *string*.

See the `cl-ppcre:scan` function, for more details.

`match-start` &optional *subexp* [Function]
Return start index of text matched by last search.

Optional argument *subexp* (a non-negative integer) specifies a parenthesized expression. A value of zero means the entire match. This is the default.

Value is nil if there is no match.

`match-end` &optional *subexp* [Function]
Return end index of text matched by last search.

Optional argument *subexp* (a non-negative integer) specifies a parenthesized expression. A value of zero means the entire match. This is the default.

Value is nil if there was no match.

`match-data` [Function]
Return list of bounding indices on what the last search matched.

`save-match-data` &body *body* [Macro]
Save match data, execute *body* forms, restore match data.

Value is the value of the last form in *body*.

`match-string` &optional *subexp* [Function]
Return string of text matched by last search.

Optional argument *subexp* (a non-negative integer) specifies a parenthesized expression. A value of zero means the entire match. This is the default.

Value is nil if there is no match.

`match-strings` [Function]
Return strings of text matched by last search.

Value is a list. A list element of nil means that the corresponding parenthesized expression did not match.

`replace-match` *new-text* &optional *subexp* [Function]
Replace text matched by last search.

Optional argument *subexp* (a non-negative integer) specifies a parenthesized expression. A value of zero means the entire match. This is the default.

5 Programming for the Real World

This chapter documents features for interfacing with the operating system. While these are especially useful when creating standalone applications, they are also highly non-portable.

5.1 Environment Variables

`environment-variables` [Function]
Return all environment variables as an associated list. List elements are cons cells of the form

```
(name . value)
```

where *name* and *value* are the name respective value of an environment variable.

`environment-variable` *name* [Function]
Return the value of the environment variable *name*.

Value is nil if no entry with key *name* exists.

`(setf environment-variable)` *value name* &optional *replace* [Function]
Set the value of the environment variable *name* to *value*.

If the environment already contains an entry with key *name* and optional argument *replace* is true (this is the default), replace the entry with key *name*. Otherwise, do nothing.

If *value* is nil, remove the entry with key *name* from the environment.

5.2 Program Arguments

`program-invocation-name` [Function]
Return the program name as invoked on the command line.

Value is a string.

`(setf program-invocation-name)` *value* [Function]
Set the program name.

Value has to be a string, a pathname, or a file stream.

`program-invocation-short-name` [Function]
Return the program name as invoked on the command line but without the directory part.

Value is a string.

`program-arguments` [Function]
Return the list of program arguments.

Value is a list of strings.

`(setf program-arguments)` *value* [Function]
Set the list of program arguments.

Value has to be a list of strings.

5.3 Program Termination

`exit-success` [Function]
Terminate the program indicating successful completion.

`exit-failure` [Function]
Terminate the program indicating a failure condition.

5.4 Diagnostic Messages

Standalone programs should write diagnostic messages to standard error. Diagnostic messages have the form

program-name:*file-name*:*line-number*: *message*

where *program-name* is the name of the program issuing the diagnostic message and *file-name* and *line-number* should point to the location of the error.

`diagnostic-message` (`simple-condition`) [Condition]
Condition type for a diagnostic message.

- Slot *program-name* is the program name. Default is the value returned by the `program-invocation-short-name` function. If nil, the program name is not part of the message text.
- Slot *file-name* is the file name operated on. Default nil, that means not applicable.
- Slot *line-number* is the line number operated on. Default nil, that means not applicable. Value is only used if *file-name* is not null.
- Slot *level* is the severity level. Value is either `:error`, `:warning`, or `:message`. Default is `:message`.

`say` *datum* &rest *arguments* [Function]
Signal a condition.

Argument is a condition designator.

If the condition is not handled, print the condition report to the `*error-output*` stream and return the condition object. Otherwise, the value is nil.

`die` *datum* &rest *arguments* [Function]
Signal a fatal condition.

Argument is a condition designator.

If the condition is not handled, print the condition report to the `*error-output*` stream and terminate the program (see function [`exit-failure`, page 15]).

You can call `say` and `die` in various ways.

```
(say (make-condition 'diagnostic-message
                     :file-name file-name
                     :format-control "no such file"))
```

This is exactly equal to the form

```
(say 'diagnostic-message
     :file-name file-name
     :format-control "no such file")
```

The most simple form is by passing the format control string as the first argument.

```
(let ((n 2))
  (when (oddp n)
    (die "should not happen")))
```

`standalone-program-p` [Function]
Return true if Lisp is running in batch mode.

`standalone-program` [Function]
Disable features available in an interactive Lisp.

5.5 Working Directory

`get-working-directory` [Function]
Return the process's working directory.

Value is a pathname.

Signal a `file-error` if the directory can not be determined.

`set-working-directory` *directory* &optional *default* [Function]
Set the process's working directory to *directory*.

Value is the pathname of the new working directory.

- First argument *directory* is either a string (interpreted as a directory file name) or a pathspec.
- If optional second argument *default* is true, adjust the special variable `*default-pathname-defaults*`, too. This is the default.

Signal a `file-error` if the directory can not be changed.

`with-working-directory` (*directory* &rest *arg*) &body *body* [Macro]
Temporarily change the process' working directory to *directory* and evaluate *body*.

5.6 Temporary Files and Directories

`temporary-file-name` &key *prefix directory* [Function]
Generate a pathname that may be used for a temporary file.

- Keyword argument *prefix* is the initial sequence of characters for the file name. Default is `"temp"`.
- Keyword argument *directory* specifies the directory in which the file name is created. Value is either nil, t, a string, or a pathname designator. A value of nil means to utilize the directory part of `*default-pathname-defaults*`, t means to use some system specific temporary directory, a string is interpreted as a directory file name, and a pathname designator is used as is.

Value is a pathname whose file name is *prefix* followed by six random characters (see the `random-string` function).

When keyword argument *directory* is t, the system specific temporary directory is chosen as follows. First, the environment variables `TMPDIR` and `TMP` are examined in that order. If both environment variables are not set, further processing depends on the type of operating system. On Unix, fall back to the `/tmp` directory. On Windows, check the value of the environment variable `TEMP`, then fall back to the `C:\Temp`, directory.

`temporary-file` &key *prefix directory direction element-type external-format* [Function]
Create a unique file.

- Keyword arguments *prefix* and *directory* have the same meaning as for the `temporary-file-name` function.
- Keyword arguments *direction* (default `:output`), *element-type* (default `character`), and *external-format* (default `:default`) have the same meaning as for the `open` function.

Value is a file stream to the newly created file.

`with-temporary-file` (*var* &rest *arg*) &body *body* [Macro]
Create a temporary file and evaluate the body forms.

- First argument *var* is the variable name to which the file stream of the temporary file is bound.

- Remaining arguments are passed on to the `temporary-file` function.

When control leaves the body, either normally or abnormally, the temporary file is automatically closed and deleted.

Value is the value of the last form of *body*.

`temporary-directory` &key *prefix directory* [Function]
Create a unique directory.

- Keyword arguments *prefix* and *directory* have the same meaning as for the `temporary-file-name` function.

Value is the pathname to the newly created directory.

`with-temporary-directory` (*var* &rest *arg*) &body *body* [Macro]
Create a temporary directory and evaluate the body forms.

- First argument *var* is the variable name to which the pathname of the temporary directory is bound.
- Remaining arguments are passed on to the `temporary-directory` function.

When control leaves the body, either normally or abnormally, the temporary directory is automatically deleted. This includes all files and directories within the temporary directory.

Value is the value of the last form of *body*.

5.7 External Programs

`execute-program` *program* &optional *arguments* &key *extra-arguments input* [Function]
if-input-does-not-exist output if-output-exists error if-error-exists wait
Run an external program.

- First argument *program* is the program file name. Value is either a string or a pathname. If *program* is an absolute or explicit relative file name, execute the specified file. Otherwise, search for it in the standard program search path.
- Second argument *arguments* are the program arguments. Value is a list of strings.
- Keyword argument *extra-arguments* are additional program arguments. Value is a list of strings. These arguments are appended to the normal *arguments* in a way similar to the `xargs` utility. Use of this keyword may result in multiple invocations of *program*.
- Keyword argument *input* is the source for the program's standard input stream. Value is either nil, t, `:stream`, a string, or a pathname. Default is t and `:stream` is only valid if *wait* is nil.
- Keyword argument *if-input-does-not-exist* specifies what to do if *input* names a nonexisting file. Value is either `:error`, `:create`, or nil. Default is `:error`.
- Keyword argument *output* is the destination for the program's standard output stream. Value is either nil, t, `:stream`, a string, or a pathname. Default is t and `:stream` is only valid if *wait* is nil.
- Keyword argument *if-output-exists* specifies what to do if *output* names an existing file. Value is either `:error`, `:supersede`, `:append`, or nil. Default is `:error`.
- Keyword argument *error* is the destination for the program's standard error stream. Value is either nil, t, `:stream`, a string, or a pathname. Default is t and `:stream` is only valid if `wait` is nil.
- Keyword argument *if-error-exists* specifies what to do if *error* names am existing file. Value is either `:error`, `:supersede`, `:append`, or nil. Default is `:error`.

- If keyword argument *wait* is true, block the Lisp process and wait for the program to terminate. Otherwise run the program asynchronously. Default is true.

If *input* names a non-existing file and *if-input-does-not-exist* is nil, value is nil (no error). Likewise if *output/error* names an existing file and *if-output-exists/if-error-exists* is nil. Otherwise, the return value depends on the *wait* flag. If *wait* is true, value is the program's exit status. Otherwise, value is an object representing the external program.

For *input*, *output*, and *error*, a value of nil means that the stream is redirected to the null device, t means to inherit the stream from the Lisp process, `:stream` means to create a new stream (only valid if `wait` is nil), and a string or a pathname names a file.

For *if-input-does-not-exist*, a value of `:error` means to signal a file error, `:create` means to create the file, and nil means to fail silently.

For *if-output-exists* and *if-error-exists*, a value of `:error` means to signal a file error, `:supersede` means to create a new file with the same name, `:append` means to modify the existing file at the end, and nil means to fail silently.

`with-input-from-program` (*var program* &optional *arguments* &key *input* *if-input-does-not-exist error if-error-exists*) &body *body* [Macro]

Run an external program asynchronously and evaluate the body forms.

- First argument *var* is the variable name to which the output stream of the external program is bound (see `program-output-stream`). The Lisp process can read the program's output via this stream.
- Remaining arguments have the same meaning as for the `execute-program` function.

When control leaves the body, either normally or abnormally, the open stream is automatically closed and deleted.

Value is the value of the last form of *body*.

`with-output-to-program` (*var program* &optional *arguments* &key *output* *if-output-exists error if-error-exists*) &body *body* [Macro]

Run an external program asynchronously and evaluate the body forms.

- First argument *var* is the variable name to which the input stream of the external program is bound (see `program-input-stream`). The Lisp process can write the program's input via this stream.
- Remaining arguments have the same meaning as for the `execute-program` function.

When control leaves the body, either normally or abnormally, the open stream is automatically closed and deleted.

Value is the value of the last form of *body*.

`with-program-io` (*var program* &optional *arguments* &key *error if-error-exists*) &body *body* [Macro]

Run an external program asynchronously and evaluate the body forms.

- First argument *var* is the variable name to which a bidirectional input/output stream of the external program is bound. The Lisp process can read/write the program's output/input via this stream.
- Remaining arguments have the same meaning as for the `execute-program` function.

When control leaves the body, either normally or abnormally, the open stream is automatically closed and deleted.

Value is the value of the last form of *body*.

`program-input-stream` *self* [Generic Function]
The input stream of an external program, or nil.
Argument *self* is an an object representing the external program.
The Lisp process can write to the program's input stream.

`program-output-stream` *self* [Generic Function]
The output stream of an external program, or nil.
Argument *self* is an an object representing the external program.
The Lisp process can read from the program's output stream.

`program-error-stream` *self* [Generic Function]
The error stream of an external program, or nil.
Argument *self* is an an object representing the external program.
The Lisp process can read from the program's error stream.

`program-exit-status` *self* [Generic Function]
The exit value of an external program or the negative signal value, or nil.
Argument *self* is an an object representing the external program.

`close-program-streams` *self* [Function]
Close all streams of an external program.
Argument *self* is an an object representing the external program.

Symbol Index

R

S

T

W

Concept Index

(Index is nonexistent)