UIOP Manual

Table of Contents

1 Introduction

UIOP is the portability layer of ASDF. It provides utilities that abstract over discrepancies between implementations, between operating systems, and between what the standard provides and what programmers actually need, to write portable Common Lisp programs.

It is organized by topic in many files, each of which defines its own package according to its topic: e.g `pathname.lisp` will define package `uiop/pathname` and contain utilities related to the handling of pathname objects. All exported symbols are reexported in a convenience package `uiop`, except for those from `uiop/common-lisp`. We recommend package `uiop` be used to access all the symbols.

The following API reference is auto-generated from the docstrings in the code. The chapters are arranged in dependency order.

2 UIOP/PACKAGE

`find-package*` *package-designator &optional errorp* [Function]
Like `cl:find-package`, but by default raises a `uiop:no-such-package-error` if the package is not found.

`find-symbol*` *name package-designator &optional error* [Function]
Find a symbol in a package of given string'ified `name`; unlike `cl:find-symbol`, work well with 'modern' case sensitive syntax by letting you supply a symbol or keyword for the name; also works well when the package is not present. If optional `error` argument is `nil`, return `nil` instead of an error when the symbol is not found.

`rehome-symbol` *symbol package-designator* [Function]
Changes the home package of a symbol, also leaving it present in its old home if any

`symbol-call` *package name &rest args* [Function]
Call a function associated with symbol of given name in given package, with given `args`. Useful when the call is read before the package is loaded, or when loading the package is optional.

`define-package` *package &rest clauses* [Macro]
`define-package` takes a `package` and a number of `clauses`, of the form (`keyword . args`). `define-package` supports the following keywords: `shadow`, `shadowing-import-from`, `import-from`, `export`, `intern`, `nicknames`, `documentation` -- as per `cl:defpackage`. `use` -- as per `cl:defpackage`, but if neither `use`, `use-reexport`, `mix`, nor `mix-reexport` is supplied, then it is equivalent to specifying (`:use :common-lisp`). This is unlike `cl:defpackage` for which the behavior of a form without `use` is implementation-dependent. `recycle` -- Recycle the package's exported symbols from the specified packages, in order. For every symbol scheduled to be exported by the `define-package`, either through an `:export` option or a `:reexport` option, if the symbol exists in one of the `:recycle` packages, the first such symbol is re-homed to the package being defined. For the sake of idempotence, it is important that the package being defined should appear in first position if it already exists, and even if it doesn't, ahead of any package that is not going to be deleted afterwards and never created again. In short, except for special cases, always make it the first package on the list if the list is not empty. `mix` -- Takes a list of package designators. `mix` behaves like (`:use pkg1 pkg2` ... PKGn) but additionally uses `:shadowing-import-from` to resolve conflicts in favor of the first found symbol. It may still yield an error if there is a conflict with an explicitly `:import-from` symbol. `reexport` -- Takes a list of package designators. For each package, p, in the list, export symbols with the same name as those exported from p. Note that in the case of shadowing, etc. the symbols with the same name may not be the same symbols. `unintern` -- Remove symbols here from `package`. Note that this is primarily useful when *redefining* a previously-existing package in the current image (e.g., when upgrading ASDF). Most programmers will have no use for this option. `local-nicknames` -- If the host implementation supports package local nicknames (check for the `:package-local-nicknames` feature), then this should be a list of nickname and package name pairs. Using this option will cause an error if

the host CL implementation does not support it. `use-reexport`, `mix-reexport` -- Use or mix the specified packages as per the `use` or `mix` directives, and reexport their contents as per the `reexport` directive.

3 UIOP/COMMON-LISP

`uiop/common-lisp` lets you paper over various sub-standard implementations.

This package reexports all the symbols in `common-lisp` package.

4 UIOP/UTILITY

`access-at-count` *at* [Function]
: From an `at` specification, extract a `count` of maximum number of sub-objects to read as per `access-at`

`access-at` *object at* [Function]
: Given an `object` and an `at` specifier, list of successive accessors, call each accessor on the result of the previous calls. An accessor may be an integer, meaning a call to `elt`, a keyword, meaning a call to `getf`, `nil`, meaning identity, a function or other symbol, meaning itself, or a list of a function designator and arguments, interpreted as per `ensure-function`. As a degenerate case, the `at` specifier may be an atom of a single such accessor instead of a list.

`base-string-p` *string* [Function]
: Does the `string` only contain BASE-CHARs?

`boolean-to-feature-expression` *value* [Function]
: Converts a boolean `value` to a form suitable for testing with `#+`.

`call-function` *function-spec &rest arguments* [Function]
: Call the function designated by `function-spec` as per `ensure-function`, with the given `arguments`

`call-functions` *function-specs* [Function]
: For each function in the list `function-specs`, in order, call the function as per `call-function`

`call-with-muffled-conditions` *thunk conditions* [Function]
: calls the `thunk` in a context where the `conditions` are muffled

`coerce-class` *class &key package super error* [Function]
: Coerce `class` to a class that is subclass of `super` if specified, or invoke `error` handler as per `call-function`.

  A keyword designates the name a symbol, which when found in either `package`, designates a class. `--` for backward compatibility, `*package*` is also accepted for now, but this may go in the future. A string is read as a symbol while in `package`, the symbol designates a class.

  A class object designates itself. `nil` designates itself (no class). A symbol otherwise designates a class by name.

`emptyp` *x* [Function]
: Predicate that is true for an empty sequence

`ensure-function` *fun &key package* [Function]
: Coerce the object `fun` into a function.

  If `fun` is a `function`, return it. If the `fun` is a non-sequence literal constant, return constantly that, i.e. for a boolean keyword character number or pathname. Otherwise if `fun` is a non-literally constant symbol, return its `fdefinition`. If `fun` is a `cons`,

return the function that applies its `car` to the appended list of the rest of its `cdr` and the arguments, unless the `car` is `lambda`, in which case the expression is evaluated. If `fun` is a string, `read` a form from it in the specified `package` (default: CL) and `eval` that in a (`function` ...) context.

`ensure-gethash` *key table default* [Function]
Lookup the `table` for a `key` as by `gethash`, but if not present, call the (possibly constant) function designated by `default` as per `call-function`, set the corresponding entry to the result in the table. Return two values: the entry after its optional computation, and whether it was found

`find-standard-case-symbol` *name-designator package-designator &optional error* [Function]
Find a symbol designated by `name-designator` in a package designated by `package-designator`, where `standard-case-symbol-name` is used to transform them if these designators are strings. If optional `error` argument is `nil`, return `nil` instead of an error when the symbol is not found.

`first-char` *s* [Function]
Return the first character of a non-empty string `s`, or `nil`

`frob-substrings` *string substrings &optional frob* [Function]
for each substring in `substrings`, find occurrences of it within `string` that don't use parts of matched occurrences of previous strings, and `frob` them, that is to say, remove them if `frob` is `nil`, replace by `frob` if `frob` is a `string`, or if `frob` is a `function`, call `frob` with the match and a function that emits a string in the output. Return a string made of the parts not omitted or emitted by `frob`.

`last-char` *s* [Function]
Return the last character of a non-empty string `s`, or `nil`

`lexicographic<=` *element< x y* [Function]
Lexicographically compare two lists of using the function element< to compare elements. element< is a strict total order; the resulting order on `x` and `y` will be a non-strict total order.

`lexicographic<` *element< x y* [Function]
Lexicographically compare two lists of using the function element< to compare elements. element< is a strict total order; the resulting order on `x` and `y` will also be strict.

`list-to-hash-set` *list* **&aux** (*h* **(make-hash-table test (quote equal))**) [Function]
Convert a `list` into hash-table that has the same elements when viewed as a set, up to the given equality `test`

`load-uiop-debug-utility` *&key package utility-file* [Function]
Load the `uiop` debug utility in given `package` (default `*package*`). Beware: The utility is located by `eval`'uating the `utility-file` form (default `*uiop-debug-utility*`).

`match-any-condition-p` *condition conditions* [Function]
match `condition` against any of the patterns of `conditions` supplied

`match-condition-p` *x condition* [Function]
Compare received `condition` to some pattern `x`: a symbol naming a condition class, a simple vector of length `2`, arguments to find-symbol* with result as above, or a string describing the format-control of a simple-condition.

`not-implemented-error` *functionality &optional format-control &rest format-arguments* [Function]
Signal an error because some `functionality` is not implemented in the current version of the software on the current platform; it may or may not be implemented in different combinations of version of the software and of the underlying platform. Optionally, report a formatted error message.

`parameter-error` *format-control functionality &rest format-arguments* [Function]
Signal an error because some `functionality` or its specific implementation on a given underlying platform does not accept a given parameter or combination of parameters. Report a formatted error message, that takes the functionality as its first argument (that can be skipped with `~*`).

`parse-body` *body &key documentation whole* [Function]
Parses `body` into (values remaining-forms declarations doc-string). Documentation strings are recognized only if `documentation` is true. Syntax errors in body are signalled and `whole` is used in the signal arguments when given.

`reduce/strcat` *strings &key key start end* [Function]
Reduce a list as if by `strcat`, accepting `key` `start` and `end` keywords like `reduce`. `nil` is interpreted as an empty string. A character is interpreted as a string of length one.

`register-hook-function` *variable hook &optional call-now-p* [Function]
Push the `hook` function (a designator as per `ensure-function`) onto the hook `variable`. When `call-now-p` is true, also call the function immediately.

`remove-plist-key` *key plist* [Function]
Remove a single key from a plist

`remove-plist-keys` *keys plist* [Function]
Remove a list of keys from a plist

`split-string` *string &key max separator* [Function]
Split `string` into a list of components separated by any of the characters in the sequence `separator`. If `max` is specified, then no more than max(`1`,`max`) components will be returned, starting the separation from the end, e.g. when called with arguments `"a.b.c.d.e"` :max `3` :separator `"."` it will return (`"a.b.c"` `"d"` `"e"`).

`standard-case-symbol-name` *name-designator* [Function]
Given a `name-designator` for a symbol, if it is a symbol, convert it to a string using `string`; if it is a string, use `string-upcase` on an ANSI CL platform, or `string` on a so-called `"modern"` platform such as Allegro with modern syntax.

`strcat` *&rest strings* [Function]
Concatenate strings. `nil` is interpreted as an empty string, a character as a string of length one.

`string-enclosed-p` *prefix string suffix* [Function]
Does `string` begin with `prefix` and end with `suffix`?

`string-prefix-p` *prefix string* [Function]
Does `string` begin with `prefix`?

`string-suffix-p` *string suffix* [Function]
Does `string` end with `suffix`?

`strings-common-element-type` *strings* [Function]
What least subtype of `character` can contain all the elements of all the `strings`?

`stripln` *x* [Function]
Strip a string `x` from any ending CR, LF or CRLF. Return two values, the stripped string and the ending that was stripped, or the original value and `nil` if no stripping took place. Since our `strcat` accepts `nil` as empty string designator, the two results passed to `strcat` always reconstitute the original string

`symbol-test-to-feature-expression` *name package* [Function]
Check if a symbol with a given `name` exists in `package` and returns a form suitable for testing with `#+`.

`appendf` *place &rest args* [Macro]
Append onto list

`nest` *&rest things* [Macro]
Macro to keep code nesting and indentation under control.

`uiop-debug` *&rest keys* [Macro]
Load the `uiop` debug utility at compile-time as well as runtime

`while-collecting` (**&rest** *collectors*) *&body body* [Macro]
`collectors` should be a list of names for collections. A collector defines a function that, when applied to an argument inside `body`, will add its argument to the corresponding collection. Returns multiple values, a list for each collection, in order. `e`.g., (while-collecting (foo bar) (dolist (x '((a `1`) (b `2`) (c `3`))) (foo (first x)) (bar (second x)))) Returns two values: (A `b` `c`) and (`1` `2` `3`).

`with-muffled-conditions` (*conditions*) *&body body* [Macro]
Shorthand syntax for `call-with-muffled-conditions`

`with-upgradability` (**&optional**) *&body body* [Macro]
Evaluate `body` at compile- load- and run- times, with `defun` and `defgeneric` modified to also declare the functions `notinline` and to accept a wrapping the function name specification into a list with keyword argument `supersede` (which defaults to `t` if the name is not wrapped, and `nil` if it is wrapped). If `supersede` is true, call `undefine-function` to supersede any previous definition.

`*uiop-debug-utility*` [Variable]

form that evaluates to the pathname to your favorite debugging utilities

5 UIOP/VERSION

`next-version` *version* [Function]
: When `version` is not nil, it is a string, then parse it as a version, compute the next version and return it as a string.

`parse-version` *version-string &optional on-error* [Function]
: Parse a `version-string` as a series of natural numbers separated by dots. Return a (non-null) list of integers if the string is valid; otherwise return `nil`.

  When invalid, `on-error` is called as per `call-function` before to return `nil`, with format arguments explaining why the version is invalid. `on-error` is also called if the version is not canonical in that it doesn't print back to itself, but the list is returned anyway.

`unparse-version` *version-list* [Function]
: From a parsed version (a list of natural numbers), compute the version string

`version-deprecation` *version &key style-warning warning error delete* [Function]
: Given a `version` string, and the starting versions for notifying the programmer of various levels of deprecation, return the current level of deprecation as per `with-deprecation` that is the highest level that has a declared version older than the specified version. Each start version for a level of deprecation can be specified by a keyword argument, or if left unspecified, will be the `next-version` of the immediate lower level of deprecation.

`version=` *version1 version2* [Function]
: Given two version strings, return `t` if the first is newer or the same and the second is also newer or the same.

`version<=` *version1 version2* [Function]
: Given two version strings, return `t` if the second is newer or the same

`version<` *version1 version2* [Function]
: Given two version strings, return `t` if the second is strictly newer

`with-deprecation` (*level*) *&body definitions* [Macro]
: Given a deprecation `level` (a form to be `eval`'ed at macro-expansion time), instrument the `defun` and `defmethod` forms in `definitions` to notify the programmer of the deprecation of the function when it is compiled or called.

  Increasing levels (as result from evaluating `level`) are: `nil` (not deprecated yet), `:style-warning` (a style warning is issued when used), `:warning` (a full warning is issued when used), `:error` (a continuable error instead), and `:delete` (it's an error if the code is still there while at that level).

  Forms other than `defun` and `defmethod` are not instrumented, and you can protect a `defun` or `defmethod` from instrumentation by enclosing it in a `progn`.

6 UIOP/OS

`architecture` [Function]
The CPU architecture of the current host

`chdir` *x* [Function]
Change current directory, as per POSIX chdir(2), to a given pathname object

`detect-os` [Function]
Detects the current operating system. Only needs be run at compile-time, except on ABCL where it might change between FASL compilation and runtime.

`featurep` *x &optional *features** [Function]
Checks whether a feature expression `x` is true with respect to the `*features*` set, as per the CLHS standard for `#+` and `#-`. Beware that just like the CLHS, we assume symbols from the `keyword` package are used, but that unless you're using #+/#- your reader will not have magically used the `keyword` package, so you need specify keywords explicitly.

`getcwd` [Function]
Get the current working directory as per POSIX getcwd(3), as a pathname object

`getenv` *x* [Function]
Query the environment, as in `c` getenv. Beware: may return empty string if a variable is present but empty; use getenvp to return `nil` in such a case.

`getenvp` *x* [Function]
Predicate that is true if the named variable is present in the libc environment, then returning the non-empty string value of the variable

`hostname` [Function]
return the hostname of the current host

`implementation-identifier` [Function]
Return a string that identifies the `abi` of the current implementation, suitable for use as a directory name to segregate Lisp FASLs, `c` dynamic libraries, etc.

`implementation-type` [Function]
The type of Lisp implementation used, as a short UIOP-standardized keyword

`lisp-version-string` [Function]
return a string that identifies the current Lisp implementation version

`operating-system` [Function]
The operating system of the current host

`os-genera-p` [Function]
Is the underlying operating system Genera (running on a Symbolics Lisp Machine)?

`os-macosx-p` [Function]
Is the underlying operating system MacOS `x`?

`os-unix-p` [Function]
Is the underlying operating system some Unix variant?

`os-windows-p` [Function]
Is the underlying operating system Microsoft Windows?

`parse-file-location-info` *s* [Function]
helper to parse-windows-shortcut

`parse-windows-shortcut` *pathname* [Function]
From a .lnk windows shortcut, extract the pathname linked to

`read-little-endian` *s &optional bytes* [Function]
Read a number in little-endian format from an byte (octet) stream `s`, the number having `bytes` octets (defaulting to `4`).

`read-null-terminated-string` *s* [Function]
Read a null-terminated string from an octet stream `s`

`getenv` *x* [Setf Expander]
Set an environment variable.

`*implementation-type*` [Variable]
The type of Lisp implementation used, as a short UIOP-standardized keyword

7 UIOP/PATHNAME

`absolute-pathname-p` *pathspec* [Function]
: If `pathspec` is a pathname or namestring object that parses as a pathname possessing an `:absolute` directory component, return the (parsed) pathname. Otherwise return `nil`

`call-with-enough-pathname` *maybe-subpath defaults-pathname thunk* [Function]
: In a context where `*default-pathname-defaults*` is bound to `defaults-pathname` (if not null, or else to its current value), call `thunk` with `enough-pathname` for `maybe-subpath` given `defaults-pathname` as a base pathname.

`denormalize-pathname-directory-component` *directory-component* [Function]
: Convert the `directory-component` from a CLHS-standard format to a format usable by the underlying implementation's `make-pathname` and other primitives

`directorize-pathname-host-device` *pathname* [Function]
: Given a `pathname`, return a pathname that has representations of its `host` and `device` components added to its `directory` component. This is useful for output translations.

`directory-pathname-p` *pathname* [Function]
: Does `pathname` represent a directory?

  A directory-pathname is a pathname _without_ a filename. The three ways that the filename components can be missing are for it to be `nil`, `:unspecific` or the empty string.

  Note that this does _not_ check to see that `pathname` points to an actually-existing directory.

`directory-separator-for-host` *&optional pathname* [Function]
: Given a `pathname`, return the character used to delimit directory names on this host and device.

`enough-pathname` *maybe-subpath base-pathname* [Function]
: if `maybe-subpath` is a pathname that is under `base-pathname`, return a pathname object that when used with `merge-pathnames*` with defaults `base-pathname`, returns `maybe-subpath`.

`ensure-absolute-pathname` *path &optional defaults on-error* [Function]
: Given a pathname designator `path`, return an absolute pathname as specified by `path` considering the `defaults`, or, if not possible, use `call-function` on the specified `on-error` behavior, with a format control-string and other arguments as arguments

`ensure-directory-pathname` *pathspec &optional on-error* [Function]
: Converts the non-wild pathname designator `pathspec` to directory form.

`ensure-pathname` *pathname &key on-error defaults type dot-dot namestring empty-is-nil want-pathname want-logical want-physical ensure-physical want-relative want-absolute ensure-absolute ensure-subpath want-non-wild want-wild wilden want-file want-directory ensure-directory want-existing ensure-directories-exist truename resolve-symlinks truenamize* **&aux** *p* [Function]

Coerces its argument into a `pathname`, optionally doing some transformations and checking specified constraints.

If the argument is `nil`, then `nil` is returned unless the `want-pathname` constraint is specified.

If the argument is a `string`, it is first converted to a pathname via `parse-unix-namestring`, `parse-namestring` or `parse-native-namestring` respectively depending on the `namestring` argument being `:unix`, `:lisp` or `:native` respectively, or else by using `call-function` on the `namestring` argument; if `:unix` is specified (or `nil`, the default, which specifies the same thing), then `parse-unix-namestring` it is called with the keywords `defaults type dot-dot ensure-directory want-relative`, and the result is optionally merged into the `defaults` if `ensure-absolute` is true.

The pathname passed or resulting from parsing the string is then subjected to all the checks and transformations below are run.

Each non-nil constraint argument can be one of the symbols `t`, `error`, `cerror` or `ignore`. The boolean `t` is an alias for `error`. `error` means that an error will be raised if the constraint is not satisfied. `cerror` means that an continuable error will be raised if the constraint is not satisfied. `ignore` means just return `nil` instead of the pathname.

The `on-error` argument, if not `nil`, is a function designator (as per `call-function`) that will be called with the the following arguments: a generic format string for ensure pathname, the pathname, the keyword argument corresponding to the failed check or transformation, a format string for the reason `ensure-pathname` failed, and a list with arguments to that format string. If `on-error` is `nil`, `error` is used instead, which does the right thing. You could also pass (`cerror "continue despite failed check"`).

The transformations and constraint checks are done in this order, which is also the order in the lambda-list:

`empty-is-nil` returns `nil` if the argument is an empty string. `want-pathname` checks that pathname (after parsing if needed) is not null. Otherwise, if the pathname is `nil`, ensure-pathname returns `nil`. `want-logical` checks that pathname is a `logical-pathname want-physical` checks that pathname is not a `logical-pathname ensure-physical` ensures that pathname is physical via `translate-logical-pathname want-relative` checks that pathname has a relative directory component `want-absolute` checks that pathname does have an absolute directory component `ensure-absolute` merges with the `defaults`, then checks again that the result absolute is an absolute pathname indeed. `ensure-subpath` checks that the pathname is a subpath of the `defaults`. `want-file` checks that pathname has a non-nil `file` component `want-directory` checks that pathname has nil `file` and `type` components `ensure-directory` uses `ensure-directory-pathname` to

interpret any file and type components as being actually a last directory component. `want-non-wild` checks that pathname is not a wild pathname `want-wild` checks that pathname is a wild pathname `wilden` merges the pathname with **/*.*.* if it is not wild `want-existing` checks that a file (or directory) exists with that pathname. `ensure-directories-exist` creates any parent directory with `ensure-directories-exist`. `truename` replaces the pathname by its truename, or errors if not possible. `resolve-symlinks` replaces the pathname by a variant with symlinks resolved by `resolve-symlinks`. `truenamize` uses `truenamize` to resolve as many symlinks as possible.

`file-pathname-p` *pathname* [Function]
Does `pathname` represent a file, i.e. has a non-null `name` component?

Accepts `nil`, a string (converted through `parse-namestring`) or a `pathname`.

Note that this does _not_ check to see that `pathname` points to an actually-existing file.

Returns the (parsed) `pathname` when true

`hidden-pathname-p` *pathname* [Function]
Return a boolean that is true if the pathname is hidden as per Unix style, i.e. its name starts with a dot.

`logical-pathname-p` *x* [Function]
is `x` a logical-pathname?

`make-pathname-component-logical` *x* [Function]
Make a pathname component suitable for use in a logical-pathname

`make-pathname-logical` *pathname host* [Function]
Take a `pathname`'s directory, name, type and version components, and make a new pathname with corresponding components and specified logical `host`

`make-pathname*` *&rest keys &key directory host device name type version defaults* [Function]
Takes arguments like `cl:make-pathname` in the CLHS, and tries hard to make a pathname that will actually behave as documented, despite the peculiarities of each implementation. `deprecated:` just use `make-pathname`.

`merge-pathname-directory-components` *specified defaults* [Function]
Helper for `merge-pathnames*` that handles directory components

`merge-pathnames*` *specified &optional defaults* [Function]
`merge-pathnames*` is like `merge-pathnames` except that if the `specified` pathname does not have an absolute directory, then the `host` and `device` both come from the `defaults`, whereas if the `specified` pathname does have an absolute directory, then the `host` and `device` both come from the `specified` pathname. This is what users want on a modern Unix or Windows operating system, unlike the `merge-pathnames` behavior. Also, if either argument is `nil`, then the other argument is returned unmodified; this is unlike `merge-pathnames` which always merges with a pathname, by default `*default-pathname-defaults*`, which cannot be `nil`.

`nil-pathname` *&optional defaults* [Function]
A pathname that is as neutral as possible for use as defaults when merging, making or parsing pathnames

`normalize-pathname-directory-component` *directory* [Function]
Convert the `directory` component from a format usable by the underlying implementation's `make-pathname` and other primitives to a CLHS-standard format that is a list and not a string.

`parse-unix-namestring` *name &rest keys &key type defaults dot-dot ensure-directory &allow-other-keys* [Function]
Coerce `name` into a `pathname` using standard Unix syntax.

Unix syntax is used whether or not the underlying system is Unix; on such non-Unix systems it is reliably usable only for relative pathnames. This function is especially useful to manipulate relative pathnames portably, where it is crucial to possess a portable pathname syntax independent of the underlying OS. This is what `parse-unix-namestring` provides, and why we use it in ASDF.

When given a `pathname` object, just return it untouched. When given `nil`, just return `nil`. When given a non-null `symbol`, first downcase its name and treat it as a string. When given a `string`, portably decompose it into a pathname as below.

#\/ separates directory components.

The last #\/-separated substring is interpreted as follows: `1-` If `type` is `:directory` or `ensure-directory` is true, the string is made the last directory component, and `name` and `type` are `nil`. if the string is empty, it's the empty pathname with all slots `nil`. `2-` If `type` is `nil`, the substring is a file-namestring, and its `name` and `type` are separated by `split-name-type`. `3-` If `type` is a string, it is the given `type`, and the whole string is the `name`.

Directory components with an empty name or the name `"."` are removed. Any directory named `".."` is read as `dot-dot`, which must be one of `:back` or `:up` and defaults to `:back`.

`host`, `device` and `version` components are taken from `defaults`, which itself defaults to `*nil-pathname*`, also used if `defaults` is `nil`. No host or device can be specified in the string itself, which makes it unsuitable for absolute pathnames outside Unix.

For relative pathnames, these components (and hence the defaults) won't matter if you use `merge-pathnames*` but will matter if you use `merge-pathnames`, which is an important reason to always use `merge-pathnames*`.

Arbitrary keys are accepted, and the parse result is passed to `ensure-pathname` with those keys, removing `type defaults` and `dot-dot`. When you're manipulating pathnames that are supposed to make sense portably even though the OS may not be Unixish, we recommend you use `:want-relative t` to throw an error if the pathname is absolute

`pathname-directory-pathname` *pathname* [Function]
Returns a new pathname with same `host`, `device`, `directory` as `pathname`, and `nil` `name`, `type` and `version` components

`pathname-equal` *p1 p2* [Function]
Are the two pathnames `p1` and `p2` reasonably equal in the paths they denote?

`pathname-host-pathname` *pathname* [Function]
return a pathname with the same host as given `pathname`, and all other fields `nil`

`pathname-parent-directory-pathname` *pathname* [Function]
Returns a new pathname that corresponds to the parent of the current pathname's directory, i.e. removing one level of depth in the `directory` component. e.g. if pathname is Unix pathname /foo/bar/baz/file.type then return /foo/bar/

`pathname-root` *pathname* [Function]
return the root directory for the host and device of given `pathname`

`physical-pathname-p` *x* [Function]
is `x` a pathname that is not a logical-pathname?

`physicalize-pathname` *x* [Function]
if `x` is a logical pathname, use translate-logical-pathname on it.

`relative-pathname-p` *pathspec* [Function]
If `pathspec` is a pathname or namestring object that parses as a pathname possessing a `:relative` or `nil` directory component, return the (parsed) pathname. Otherwise return `nil`

`relativize-directory-component` *directory-component* [Function]
Given the `directory-component` of a pathname, return an otherwise similar relative directory component

`relativize-pathname-directory` *pathspec* [Function]
Given a `pathname`, return a relative pathname with otherwise the same components

`split-name-type` *filename* [Function]
Split a filename into two values `name` and `type` that are returned. We assume filename has no directory component. The last . if any separates name and type from from type, except that if there is only one . and it is in first position, the whole filename is the `name` with an empty type. `name` is always a string. For an empty type, `*unspecific-pathname-type*` is returned.

`split-unix-namestring-directory-components` *unix-namestring* [Function]
&key ensure-directory dot-dot
Splits the path string `unix-namestring`, returning four values: A flag that is either :absolute or :relative, indicating how the rest of the values are to be interpreted. A directory path --- a list of strings and keywords, suitable for use with `make-pathname` when prepended with the flag value. Directory components with an empty name or the name . are removed. Any directory named .. is read as `dot-dot`, or `:back` if it's `nil` (not `:up`). A last-component, either a file-namestring including type extension, or `nil` in the case of a directory pathname. A flag that is true iff the unix-style-pathname was just a file-namestring without / path specification. `ensure-directory` forces the namestring to be interpreted as a directory pathname: the third return value will be

`nil`, and final component of the namestring will be treated as part of the directory path.

An empty string is thus read as meaning a pathname object with all fields nil.

Note that colon characters `#:` will `not` be interpreted as host specification. Absolute pathnames are only appropriate on Unix-style systems.

The intention of this function is to support structured component names, e.g., (:file "foo/bar"), which will be unpacked to relative pathnames.

`subpathname*` *pathname subpath &key type* [Function]
returns `nil` if the base pathname is `nil`, otherwise like `subpathname`.

`subpathname` *pathname subpath &key type* [Function]
This function takes a `pathname` and a `subpath` and a `type`. If `subpath` is already a `pathname` object (not namestring), and is an absolute pathname at that, it is returned unchanged; otherwise, `subpath` is turned into a relative pathname with given `type` as per `parse-unix-namestring` with `:want-relative t :type type`, then it is merged with the `pathname-directory-pathname` of `pathname`.

`subpathp` *maybe-subpath base-pathname* [Function]
if `maybe-subpath` is a pathname that is under `base-pathname`, return a pathname object that when used with `merge-pathnames*` with defaults `base-pathname`, returns `maybe-subpath`.

`translate-pathname*` *path absolute-source destination &optional root source* [Function]
A wrapper around `translate-pathname` to be used by the ASDF output-translations facility. `path` is the pathname to be translated. `absolute-source` is an absolute pathname to use as source for translate-pathname, `destination` is either a function, to be called with `path` and `absolute-source`, or a relative pathname, to be merged with `root` and used as destination for translate-pathname or an absolute pathname, to be used as destination for translate-pathname. In that last case, if `root` is non-NIL, `path` is first transformated by `directorize-pathname-host-device`.

`unix-namestring` *pathname* [Function]
Given a non-wild `pathname`, return a Unix-style namestring for it. If the `pathname` is `nil` or a `string`, return it unchanged.

This only considers the `directory`, `name` and `type` components of the pathname. This is a portable solution for representing relative pathnames, But unless you are running on a Unix system, it is not a general solution to representing native pathnames.

An error is signaled if the argument is not `null`, a `string` or a `pathname`, or if it is a `pathname` but some of its components are not recognized.

`wilden` *path* [Function]
From a pathname, return a wildcard pathname matching any file in any subdirectory of given pathname's directory

`with-enough-pathname` (*pathname-var* **&key** *pathname defaults*) *&body body* [Macro]
Shorthand syntax for `call-with-enough-pathname`

`with-pathname-defaults` (**&optional** *defaults*) *&body body* [Macro]
Execute `body` in a context where the `*default-pathname-defaults*` is as specified, where leaving the defaults `nil` or unspecified means a (`nil-pathname`), except on ABCL, Genera and XCL, where it remains unchanged for it doubles as current-directory.

`*nil-pathname*` [Variable]
A pathname that is as neutral as possible for use as defaults when merging, making or parsing pathnames

`*output-translation-function*` [Variable]
Hook for output translations.

This function needs to be idempotent, so that actions can work whether their inputs were translated or not, which they will be if we are composing operations. e.g. if some create-lisp-op creates a lisp file from some higher-level input, you need to still be able to use compile-op on that lisp file.

`*unspecific-pathname-type*` [Variable]
Unspecific type component to use with the underlying implementation's `make-pathname`

`*wild-directory*` [Variable]
A pathname object with wildcards for matching any subdirectory

`*wild-file-for-directory*` [Variable]
A pathname object with wildcards for matching any file with `directory`

`*wild-file*` [Variable]
A pathname object with wildcards for matching any file with `translate-pathname`

`*wild-inferiors*` [Variable]
A pathname object with wildcards for matching any recursive subdirectory

`*wild-path*` [Variable]
A pathname object with wildcards for matching any file in any recursive subdirectory

`*wild*` [Variable]
Wild component for use with `make-pathname`

8 UIOP/FILESYSTEM

`call-with-current-directory` *dir thunk* [Function]
call the `thunk` in a context where the current directory was changed to `dir`, if not `nil`. Note that this operation is usually `not` thread-safe.

`collect-sub*directories` *directory collectp recursep collector* [Function]
Given a `directory`, when `collectp` returns true when `call-function`'ed with the directory, call-function the `collector` function designator on the directory, and recurse each of its subdirectories on which the `recursep` returns true when `call-function`'ed with them. This function will thus let you traverse a filesystem hierarchy, superseding the functionality of `cl-fad:walk-directory`. The behavior in presence of symlinks is not portable. Use IOlib to handle such situations.

`delete-directory-tree` *directory-pathname &key validate if-does-not-exist* [Function]
Delete a directory including all its recursive contents, aka rm -rf.

To reduce the risk of infortunate mistakes, `directory-pathname` must be a physical non-wildcard directory pathname (not namestring).

If the directory does not exist, the `if-does-not-exist` argument specifies what happens: if it is `:error` (the default), an error is signaled, whereas if it is `:ignore`, nothing is done.

Furthermore, before any deletion is attempted, the `directory-pathname` must pass the validation function designated (as per `ensure-function`) by the `validate` keyword argument which in practice is thus compulsory, and validates by returning a non-NIL result. If you're suicidal or extremely confident, just use `:validate t`.

`delete-empty-directory` *directory-pathname* [Function]
Delete an empty directory

`delete-file-if-exists` *x* [Function]
Delete a file `x` if it already exists

`directory-exists-p` *x* [Function]
Is `x` the name of a directory that exists on the filesystem?

`directory-files` *directory &optional pattern* [Function]
Return a list of the files in a directory according to the `pattern`. Subdirectories should `not` be returned. `pattern` defaults to a pattern carefully chosen based on the implementation; override the default at your own risk. `directory-files` tries `not` to resolve symlinks if the implementation permits this, but the behavior in presence of symlinks is not portable. Use IOlib to handle such situations.

`directory*` *pathname-spec &rest keys &key &allow-other-keys* [Function]
Return a list of the entries in a directory by calling `directory`. Try to override the defaults to not resolving symlinks, if implementation allows.

`ensure-all-directories-exist` *pathnames* [Function]
Ensure that for every pathname in `pathnames`, we ensure its directories exist

`file-exists-p` *x* [Function]
Is `x` the name of a file that exists on the filesystem?

`filter-logical-directory-results` *directory entries merger* [Function]
If `directory` isn't a logical pathname, return `entries`. If it is, given `entries` in the `directory`, remove the entries which are physical yet when transformed by `merger` have a different `truename`. Also remove duplicates as may appear with some translation rules. This function is used as a helper to `directory-files` to avoid invalid entries when using logical-pathnames.

`get-pathname-defaults` *&optional defaults* [Function]
Find the actual `defaults` to use for pathnames, including resolving them with respect to `getcwd` if the `defaults` were relative

`getenv-absolute-directories` *x* [Function]
Extract a list of absolute directories from a user-configured environment variable, as per native OS. Any empty entries in the environment variable `x` will be returned as NILs.

`getenv-absolute-directory` *x* [Function]
Extract an absolute directory pathname from a user-configured environment variable, as per native OS

`getenv-pathname` *x &rest constraints &key ensure-directory want-directory on-error &allow-other-keys* [Function]
Extract a pathname from a user-configured environment variable, as per native OS, check constraints and normalize as per `ensure-pathname`.

`getenv-pathnames` *x &rest constraints &key on-error &allow-other-keys* [Function]
Extract a list of pathname from a user-configured environment variable, as per native OS, check constraints and normalize each one as per `ensure-pathname`. Any empty entries in the environment variable `x` will be returned as NILs.

`inter-directory-separator` [Function]
What character does the current OS conventionally uses to separate directories?

`lisp-implementation-directory` *&key truename* [Function]
Where are the system files of the current installation of the CL implementation?

`lisp-implementation-pathname-p` *pathname* [Function]
Is the `pathname` under the current installation of the CL implementation?

`native-namestring` *x* [Function]
From a non-wildcard CL pathname, a return namestring suitable for passing to the operating system

`parse-native-namestring` *string &rest constraints &key ensure-directory &allow-other-keys* [Function]
From a native namestring suitable for use by the operating system, return a CL pathname satisfying all the specified constraints as per `ensure-pathname`

`probe-file*` *p &key truename* [Function]
: when given a pathname `p` (designated by a string as per `parse-namestring`), probes the filesystem for a file or directory with given pathname. If it exists, return its truename if `truename` is true, or the original (parsed) pathname if it is false (the default).

`rename-file-overwriting-target` *source target* [Function]
: Rename a file, overwriting any previous file with the `target` name, in an atomic way if the implementation allows.

`resolve-symlinks*` *path* [Function]
: `resolve-symlinks` in `path` iff `*resolve-symlinks*` is `t` (the default).

`resolve-symlinks` *path* [Function]
: Do a best effort at resolving symlinks in `path`, returning a partially or totally resolved `path`.

`safe-file-write-date` *pathname* [Function]
: Safe variant of `file-write-date` that may return `nil` rather than raise an error.

`split-native-pathnames-string` *string &rest constraints &key &allow-other-keys* [Function]
: Given a string of pathnames specified in native OS syntax, separate them in a list, check constraints and normalize each one as per `ensure-pathname`, where an empty string denotes `nil`.

`subdirectories` *directory* [Function]
: Given a `directory` pathname designator, return a list of the subdirectories under it. The behavior in presence of symlinks is not portable. Use IOlib to handle such situations.

`truename*` *p* [Function]
: Nicer variant of `truename` that plays well with `nil`, avoids logical pathname contexts, and tries both files and directories

`truenamize` *pathname* [Function]
: Resolve as much of a pathname as possible

`with-current-directory` (**&optional** *dir*) *&body body* [Macro]
: Call `body` while the POSIX current working directory is set to `dir`

`*resolve-symlinks*` [Variable]
: Determine whether or not ASDF resolves symlinks when defining systems. Defaults to `t`.

9 UIOP/STREAM

`add-pathname-suffix` *pathname suffix &rest keys* [Function]

Add a `suffix` to the name of a `pathname`, return a new pathname. Further `keys` can be passed to `make-pathname`.

`always-default-encoding` *pathname* [Function]

Trivial function to use as *encoding-detection-hook*, always 'detects' the *default-encoding*

`call-with-input-file` *pathname thunk &key element-type external-format if-does-not-exist* [Function]

Open `file` for input with given recognizes options, call `thunk` with the resulting stream. Other keys are accepted but discarded.

`call-with-null-input` *fun &key element-type external-format if-does-not-exist* [Function]

Call `fun` with an input stream that always returns end of file. The keyword arguments are allowed for backward compatibility, but are ignored.

`call-with-null-output` *fun &key element-type external-format if-exists if-does-not-exist* [Function]

Call `fun` with an output stream that discards all output. The keyword arguments are allowed for backward compatibility, but are ignored.

`call-with-output-file` *pathname thunk &key element-type external-format if-exists if-does-not-exist* [Function]

Open `file` for input with given recognizes options, call `thunk` with the resulting stream. Other keys are accepted but discarded.

`call-with-staging-pathname` *pathname fun* [Function]

Calls `fun` with a staging pathname, and atomically renames the staging pathname to the `pathname` in the end. `nb:` this protects only against failure of the program, not against concurrent attempts. For the latter case, we ought pick a random suffix and atomically open it.

`call-with-temporary-file` *thunk &key want-stream-p want-pathname-p direction keep after directory type prefix suffix element-type external-format* [Function]

Call a `thunk` with stream and/or pathname arguments identifying a temporary file.

The temporary file's pathname will be based on concatenating `prefix` (or `"tmp"` if it's `nil`), a random alphanumeric string, and optional `suffix` (defaults to `"-tmp"` if a type was provided) and `type` (defaults to `"tmp"`, using a dot as separator if not `nil`), within `directory` (defaulting to the `temporary-directory`) if the `prefix` isn't absolute.

The file will be open with specified `direction` (defaults to `:io`), `element-type` (defaults to `*default-stream-element-type*`) and `external-format` (defaults to `*utf-8-external-format*`). If `want-stream-p` is true (the defaults to `t`),

then `thunk` will then be `call-function`'ed with the stream and the pathname (if `want-pathname-p` is true, defaults to `t`), and stream will be closed after the `thunk` exits (either normally or abnormally). If `want-stream-p` is false, then `want-pathame-p` must be true, and then `thunk` is only `call-function`'ed after the stream is closed, with the pathname as argument. Upon exit of `thunk`, the `after` thunk if defined is `call-function`'ed with the pathname as argument. If `after` is defined, its results are returned, otherwise, the results of `thunk` are returned. Finally, the file will be deleted, unless the `keep` argument when `call-function`'ed returns true.

`concatenate-files` *inputs output* [Function]
create a new `output` file the contents of which a the concatenate of the `inputs` files.

`copy-file` *input output* [Function]
Copy contents of the `input` file to the `output` file

`copy-stream-to-stream` *input output &key element-type buffer-size linewise prefix* [Function]
Copy the contents of the `input` stream into the `output` stream. If `linewise` is true, then read and copy the stream line by line, with an optional `prefix`. Otherwise, using `write-sequence` using a buffer of size `buffer-size`.

`default-encoding-external-format` *encoding* [Function]
Default, ignorant, function to transform a character `encoding` as a portable keyword to an implementation-dependent `external-format` specification. Load system `asdf-encodings` to hook in a better one.

`default-temporary-directory` [Function]
Return a default directory to use for temporary files

`detect-encoding` *pathname* [Function]
Detects the encoding of a specified file, going through user-configurable hooks

`encoding-external-format` *encoding* [Function]
Transform a portable `encoding` keyword to an implementation-dependent `external-format`, going through all the proper hooks.

`eval-input` *input* [Function]
Portably read and evaluate forms from `input`, return the last values.

`eval-thunk` *thunk* [Function]
Evaluate a `thunk` of code: If a function, `funcall` it without arguments. If a constant literal and not a sequence, return it. If a cons or a symbol, `eval` it. If a string, repeatedly read and evaluate from it, returning the last values.

`finish-outputs` *&rest streams* [Function]
Finish output on the main output streams as well as any specified one. Useful for portably flushing I/O before user input or program exit.

`format!` *stream format &rest args* [Function]
Just like format, but call finish-outputs before and after the output.

`input-string` *&optional input* [Function]
If the desired `input` is a string, return that string; otherwise slurp the `input` into a string and return that

`null-device-pathname` [Function]
Pathname to a bit bucket device that discards any information written to it and always returns `eof` when read from

`output-string` *string &optional output* [Function]
If the desired `output` is not `nil`, print the string to the output; otherwise return the string

`println` *x &optional stream* [Function]
Variant of `princ` that also calls `terpri` afterwards

`read-file-form` *file &rest keys &key at &allow-other-keys* [Function]
Open input `file` with option `keys` (except `at`), and read its contents as per `slurp-stream-form` with given `at` specifier. `beware:` be sure to use `with-safe-io-syntax`, or some variant thereof

`read-file-forms` *file &rest keys &key count &allow-other-keys* [Function]
Open input `file` with option `keys` (except `count`), and read its contents as per `slurp-stream-forms` with given `count`. If `count` is null, read to the end of the stream; if `count` is an integer, stop after `count` forms were read. `beware:` be sure to use `with-safe-io-syntax`, or some variant thereof

`read-file-line` *file &rest keys &key at &allow-other-keys* [Function]
Open input `file` with option `keys` (except `at`), and read its contents as per `slurp-stream-line` with given `at` specifier. `beware:` be sure to use `with-safe-io-syntax`, or some variant thereof

`read-file-lines` *file &rest keys* [Function]
Open `file` with option `keys`, read its contents as a list of lines `beware:` be sure to use `with-safe-io-syntax`, or some variant thereof

`read-file-string` *file &rest keys* [Function]
Open `file` with option `keys`, read its contents as a string

`safe-format!` *stream format &rest args* [Function]
Variant of `format` that is safe against both dangerous syntax configuration and errors while printing.

`safe-read-file-form` *pathname &rest keys &key package* [Function]
&allow-other-keys
Reads the specified form from the top of a file using a safe standardized syntax. Extracts the form using `read-file-form`, within an `with-safe-io-syntax` using the specified `package`.

`safe-read-file-line` *pathname &rest keys &key package* [Function]
&allow-other-keys
Reads the specified line from the top of a file using a safe standardized syntax. Extracts the line using `read-file-line`, within an `with-safe-io-syntax` using the specified `package`.

`safe-read-from-string` *string &key package eof-error-p eof-value start end preserve-whitespace* [Function]
Read from `string` using a safe syntax, as per `with-safe-io-syntax`

`setup-temporary-directory` [Function]
Configure a default temporary directory to use.

`slurp-stream-form` *input &key at* [Function]
Read the contents of the `input` stream as a list of forms, then return the `access-at` of these forms following the `at`. `at` defaults to `0`, i.e. return the first form. `at` is typically a list of integers. If `at` is `nil`, it will return all the forms in the file.

The stream will not be read beyond the Nth form, where `n` is the index specified by path, if path is either an integer or a list that starts with an integer.

`beware:` be sure to use `with-safe-io-syntax`, or some variant thereof

`slurp-stream-forms` *input &key count* [Function]
Read the contents of the `input` stream as a list of forms, and return those forms.

If `count` is null, read to the end of the stream; if `count` is an integer, stop after `count` forms were read.

`beware:` be sure to use `with-safe-io-syntax`, or some variant thereof

`slurp-stream-line` *input &key at* [Function]
Read the contents of the `input` stream as a list of lines, then return the `access-at` of that list of lines using the `at` specifier. `path` defaults to `0`, i.e. return the first line. `path` is typically an integer, or a list of an integer and a function. If `path` is `nil`, it will return all the lines in the file.

The stream will not be read beyond the Nth lines, where `n` is the index specified by path if path is either an integer or a list that starts with an integer.

`slurp-stream-lines` *input &key count* [Function]
Read the contents of the `input` stream as a list of lines, return those lines.

Note: relies on the Lisp's `read-line`, but additionally removes any remaining CR from the line-ending if the file or stream had CR+LF but Lisp only removed LF.

Read no more than `count` lines.

`slurp-stream-string` *input &key element-type stripped* [Function]
Read the contents of the `input` stream as a string

`standard-eval-thunk` *thunk &key package* [Function]
Like `eval-thunk`, but in a more standardized evaluation context.

`temporary-directory` [Function]
Return a directory to use for temporary files

`tmpize-pathname` *x* [Function]
Return a new pathname modified from `x` by adding a trivial random suffix. A new empty file with said temporary pathname is created, to ensure there is no clash with any concurrent process attempting the same thing.

`writeln` *x &rest keys &key stream &allow-other-keys* [Function]
Variant of `write` that also calls `terpri` afterwards

`with-input` (*input-var* **&optional** *value*) *&body body* [Macro]
Bind `input-var` to an input stream, coercing `value` (default: previous binding of `input-var`) as per `call-with-input`, and evaluate `body` within the scope of this binding.

`with-null-input` (*var* **&rest** *keys* **&key** *element-type external-format if-does-not-exist*) *&body body* [Macro]
Evaluate `body` in a context when `var` is bound to an input stream that always returns end of file. The keyword arguments are allowed for backward compatibility, but are ignored.

`with-null-output` (*var* **&rest** *keys* **&key** *element-type external-format if-does-not-exist if-exists*) *&body body* [Macro]
Evaluate `body` in a context when `var` is bound to an output stream that discards all output. The keyword arguments are allowed for backward compatibility, but are ignored.

`with-output` (*output-var* **&optional** *value* **&key** *element-type*) *&body body* [Macro]
Bind `output-var` to an output stream obtained from `value` (default: previous binding of `output-var`) treated as a stream designator per `call-with-output`. Evaluate `body` in the scope of this binding.

`with-safe-io-syntax` (**&key** *package*) *&body body* [Macro]
Establish safe CL reader options around the evaluation of `body`

`with-staging-pathname` (*pathname-var* **&optional** *pathname-value*) *&body body* [Macro]
Trivial syntax wrapper for `call-with-staging-pathname`

`with-temporary-file` (**&key** *stream pathname directory prefix suffix type keep direction element-type external-format*) *&body body* [Macro]
Evaluate `body` where the symbols specified by keyword arguments `stream` and `pathname` (if respectively specified) are bound corresponding to a newly created temporary file ready for I/O, as per `call-with-temporary-file`. At least one of `stream` or `pathname` must be specified. If the `stream` is not specified, it will be closed before the `body` is evaluated. If `stream` is specified, then the `:close-stream` label if it appears in the `body`, separates forms run before and after the stream is closed. The values of the last form of the `body` (not counting the separating `:close-stream`) are returned. Upon success, the `keep` form is evaluated and the file is is deleted unless it evaluates to `true`.

`*default-encoding*` [Variable]
Default encoding for source files. The default value :utf-8 is the portable thing. The legacy behavior was :default. If you (asdf:load-system :asdf-encodings) then you will have autodetection via *encoding-detection-hook* below, reading emacs-style `-*-` coding: utf-8 `-*-` specifications, and falling back to utf-8 or latin1 if nothing is specified.

`*default-stream-element-type*` [Variable]
default element-type for open (depends on the current CL implementation)

`*encoding-detection-hook*` [Variable]
Hook for an extension to define a function to automatically detect a file's encoding

`*encoding-external-format-hook*` [Variable]
Hook for an extension (e.g. `asdf-encodings`) to define a better mapping from non-default encodings to and implementation-defined external-format's

`*stderr*` [Variable]
the original error output stream at startup

`*stdin*` [Variable]
the original standard input stream at startup

`*stdout*` [Variable]
the original standard output stream at startup

`*temporary-directory*` [Variable]
User-configurable location for temporary files

`*utf-8-external-format*` [Variable]
Default :external-format argument to pass to `cl:open` and also `cl:load` or `cl:compile-file` to best process a `utf-8` encoded file. On modern implementations, this will decode `utf-8` code points as CL characters. On legacy implementations, it may fall back on some 8-bit encoding, with non-ASCII code points being read as several CL characters; hopefully, if done consistently, that won't affect program behavior too much.

10 UIOP/IMAGE

`argv0` [Function]
: On supported implementations (most that matter), or when invoked by a proper wrapper script, return a string that for the name with which the program was invoked, i.e. argv[0] in `c`. Otherwise, return `nil`.

`call-image-dump-hook` [Function]
: Call the hook functions registered to be run before to dump an image

`call-image-restore-hook` [Function]
: Call the hook functions registered to be run when restoring a dumped image

`call-with-fatal-condition-handler` *thunk* [Function]
: Call `thunk` in a context where fatal conditions are appropriately handled

`command-line-arguments` *&optional arguments* [Function]
: Extract user arguments from command-line invocation of current process. Assume the calling conventions of a generated script that uses `--` if we are not called from a directly executable image.

`create-image` *destination lisp-object-files &key kind output-name prologue-code epilogue-code extra-object-files prelude postlude entry-point build-args no-uiop* [Function]
: On ECL, create an executable at pathname `destination` from the specified `object-files` and options

`die` *code format &rest arguments* [Function]
: Die in error with some error message

`dump-image` *filename &key output-name executable postlude dump-hook compression* [Function]
: Dump an image of the current Lisp environment at pathname `filename`, with various options.

  First, finalize the image, by evaluating the `postlude` as per `eval-input`, then calling each of the functions in `dump-hook`, in reverse order of registration by `register-image-dump-hook`.

  If `executable` is true, create an standalone executable program that calls `restore-image` on startup.

  Pass various implementation-defined options, such as `prepend-symbols` and `purity` on CCL, or `compression` on SBCL, and `application-type` on SBCL/Windows.

`fatal-condition-p` *condition* [Function]
: Is the `condition` fatal?

`handle-fatal-condition` *condition* [Function]
: Handle a fatal `condition`: depending on whether `*lisp-interaction*` is set, enter debugger or die

`print-backtrace` *&rest keys &key stream count condition* [Function]
Print a backtrace

`print-condition-backtrace` *condition &key stream count* [Function]
Print a condition after a backtrace triggered by that condition

`quit` *&optional code finish-output* [Function]
Quits from the Lisp world, with the given exit status if provided. This is designed to abstract away the implementation specific quit forms.

`raw-command-line-arguments` [Function]
Find what the actual command line for this process was.

`raw-print-backtrace` *&key stream count condition* [Function]
Print a backtrace, directly accessing the implementation

`register-image-dump-hook` *hook &optional call-now-p* [Function]
Register a the hook function to be run before to dump an image

`register-image-restore-hook` *hook &optional call-now-p* [Function]
Regiter a hook function to be run when restoring a dumped image

`restore-image` *&key lisp-interaction restore-hook prelude entry-point if-already-restored* [Function]
From a freshly restarted Lisp image, restore the saved Lisp environment by setting appropriate variables, running various hooks, and calling any specified entry point.

If the image has already been restored or is already being restored, as per `*image-restored-p*`, call the `if-already-restored` error handler (by default, a continuable error), and do return immediately to the surrounding restore process if allowed to continue.

Then, comes the restore process itself: First, call each function in the `restore-hook`, in the order they were registered with `register-image-restore-hook`. Second, evaluate the prelude, which is often Lisp text that is read, as per `eval-input`. Third, call the `entry-point` function, if any is specified, with no argument.

The restore process happens in a `with-fatal-condition-handler`, so that if `lisp-interaction` is `nil`, any unhandled error leads to a backtrace and an exit with an error status. If `lisp-interaction` is `nil`, the process also exits when no error occurs: if neither restart nor entry function is provided, the program will exit with status `0` (success); if a function was provided, the program will exit after the function returns (if it returns), with status `0` if and only if the primary return value of result is generalized boolean true, and with status `1` if this value is `nil`.

If `lisp-interaction` is true, unhandled errors will take you to the debugger, and the result of the function will be returned rather than interpreted as a boolean designating an exit code.

`shell-boolean-exit` *x* [Function]
Quit with a return code that is `0` iff argument `x` is true

`with-fatal-condition-handler` (**&optional**) *&body body* [Macro]
Execute `body` in a context where fatal conditions are appropriately handled

`*command-line-arguments*` [Variable]
Command-line arguments

`*image-dump-hook*` [Variable]
Functions to call (in order) when before an image is dumped

`*image-dumped-p*` [Variable]
Is this a dumped image? As a standalone executable?

`*image-entry-point*` [Variable]
a function with which to restart the dumped image when execution is restored from it.

`*image-postlude*` [Variable]
a form to evaluate, or string containing forms to read and evaluate before the image dump hooks are called and before the image is dumped.

`*image-prelude*` [Variable]
a form to evaluate, or string containing forms to read and evaluate when the image is restarted, but before the entry point is called.

`*image-restore-hook*` [Variable]
Functions to call (in reverse order) when the image is restored

`*lisp-interaction*` [Variable]
Is this an interactive Lisp environment, or is it batch processing?

11 UIOP/LISP-BUILD

`call-around-hook` *hook function* [Function]
Call a `hook` around the execution of `function`

`call-with-muffled-compiler-conditions` *thunk* [Function]
Call given `thunk` in a context where uninteresting conditions and compiler conditions are muffled

`call-with-muffled-loader-conditions` *thunk* [Function]
Call given `thunk` in a context where uninteresting conditions and loader conditions are muffled

`check-deferred-warnings` *files &optional context-format context-arguments* [Function]
Given a list of `files` containing deferred warnings saved by `call-with-saved-deferred-warnings`, re-intern and raise any warnings that are still meaningful.

`check-lisp-compile-results` *output warnings-p failure-p &optional context-format context-arguments* [Function]
Given the results of `compile-file`, raise an error or warning as appropriate

`check-lisp-compile-warnings` *warnings-p failure-p &optional context-format context-arguments* [Function]
Given the warnings or failures as resulted from `compile-file` or checking deferred warnings, raise an error or warning as appropriate

`combine-fasls` *inputs output* [Function]
Combine a list of FASLs `inputs` into a single FASL `output`

`compile-file-pathname*` *input-file &rest keys &key output-file &allow-other-keys* [Function]
Variant of `compile-file-pathname` that works well with `compile-file*`

`compile-file*` *input-file &rest keys &key compile-check output-file warnings-file emit-cfasl &allow-other-keys* [Function]
This function provides a portable wrapper around `compile-file`. It ensures that the `output-file` value is only returned and the file only actually created if the compilation was successful, even though your implementation may not do that. It also checks an optional user-provided consistency function `compile-check` to determine success; it will call this function if not `nil` at the end of the compilation with the arguments sent to `compile-file*`, except with `:output-file tmp-file` where `tmp-file` is the name of a temporary output-file. It also checks two flags (with legacy british spelling from `asdf1`), `*compile-file-failure-behaviour*` and `*compile-file-warnings-behaviour*` with appropriate implementation-dependent defaults, and if a failure (respectively warnings) are reported by `compile-file`, it will consider that an error unless the respective behaviour flag is one of `:success :warn :ignore`. If `warnings-file` is defined, deferred warnings are saved to that file. On ECL or MKCL, it creates both the linkable object and loadable fasl files. On implementations that erroneously do not recognize standard keyword arguments, it will filter them appropriately.

`compile-file-type` *&rest keys* [Function]
pathname `type` for lisp FASt Loading files

`current-lisp-file-pathname` [Function]
Portably return the `pathname` of the current Lisp source file being compiled or loaded

`disable-deferred-warnings-check` [Function]
Disable the saving of deferred warnings

`enable-deferred-warnings-check` [Function]
Enable the saving of deferred warnings

`get-optimization-settings` [Function]
Get current compiler optimization settings, ready to `proclaim` again

`lispize-pathname` *input-file* [Function]
From a `input-file` pathname, return a corresponding .lisp source pathname

`load-from-string` *string* [Function]
Portably read and evaluate forms from a `string`.

`load-pathname` [Function]
Portably return the `load-pathname` of the current source file or fasl. May return a relative pathname.

`load*` *x &rest keys &key &allow-other-keys* [Function]
Portable wrapper around `load` that properly handles loading from a stream.

`proclaim-optimization-settings` [Function]
Proclaim the optimization settings in `*optimization-settings*`

`reify-deferred-warnings` [Function]
return a portable S-expression, portably readable and writeable in any Common Lisp implementation using `read` within a `with-safe-io-syntax`, that represents the warnings currently deferred by `with-compilation-unit`. One of three functions required for deferred-warnings support in ASDF.

`reify-simple-sexp` *sexp* [Function]
Given a simple `sexp`, return a representation of it as a portable `sexp`. Simple means made of symbols, numbers, characters, simple-strings, pathnames, cons cells.

`reset-deferred-warnings` [Function]
Reset the set of deferred warnings to be handled at the end of the current `with-compilation-unit`. One of three functions required for deferred-warnings support in ASDF.

`save-deferred-warnings` *warnings-file* [Function]
Save forward reference conditions so they may be issued at a latter time, possibly in a different process.

`unreify-deferred-warnings` *reified-deferred-warnings* [Function]
: given a S-expression created by `reify-deferred-warnings`, reinstantiate the corresponding deferred warnings as to be handled at the end of the current `with-compilation-unit`. Handle any warning that has been resolved already, such as an undefined function that has been defined since. One of three functions required for deferred-warnings support in ASDF.

`unreify-simple-sexp` *sexp* [Function]
: Given the portable output of `reify-simple-sexp`, return the simple `sexp` it represents

`warnings-file-p` *file &optional implementation-type* [Function]
: Is `file` a saved warnings file for the given `implementation-type`? If that given type is `nil`, use the currently configured `*warnings-file-type*` instead.

`warnings-file-type` *&optional implementation-type* [Function]
: The pathname type for warnings files on given `implementation-type`, where `nil` designates the current one

`with-muffled-compiler-conditions` (**&optional**) *&body body* [Macro]
: Trivial syntax for `call-with-muffled-compiler-conditions`

`with-muffled-loader-conditions` (**&optional**) *&body body* [Macro]
: Trivial syntax for `call-with-muffled-loader-conditions`

`with-saved-deferred-warnings` (*warnings-file* **&key** *source-namestring*) *&body body* [Macro]
: Trivial syntax for `call-with-saved-deferred-warnings`

`*base-build-directory*` [Variable]
: When set to a non-null value, it should be an absolute directory pathname, which will serve as the `*default-pathname-defaults*` around a `compile-file`, what more while the input-file is shortened if possible to `enough-pathname` relative to it. This can help you produce more deterministic output for FASLs.

`*compile-check*` [Variable]
: A hook for user-defined compile-time invariants

`*compile-file-failure-behaviour*` [Variable]
: How should ASDF react if it encounters a failure (per the ANSI spec of `compile-file`) when compiling a file, which includes any non-style-warning warning. Valid values are :error, :warn, and :ignore. Note that ASDF `always` raises an error if it fails to create an output file when compiling.

`*compile-file-warnings-behaviour*` [Variable]
: How should ASDF react if it encounters a warning when compiling a file? Valid values are :error, :warn, and :ignore.

`*optimization-settings*` [Variable]
: Optimization settings to be used by `proclaim-optimization-settings`

`*output-translation-function*` [Variable]
Hook for output translations.

This function needs to be idempotent, so that actions can work whether their inputs were translated or not, which they will be if we are composing operations. e.g. if some create-lisp-op creates a lisp file from some higher-level input, you need to still be able to use compile-op on that lisp file.

`*previous-optimization-settings*` [Variable]
Optimization settings saved by `proclaim-optimization-settings`

`*uninteresting-compiler-conditions*` [Variable]
Additional conditions that may be skipped while compiling Lisp code.

`*uninteresting-conditions*` [Variable]
Conditions that may be skipped while compiling or loading Lisp code.

`*uninteresting-loader-conditions*` [Variable]
Additional conditions that may be skipped while loading Lisp code.

`*usual-uninteresting-conditions*` [Variable]
A suggested value to which to set or bind *uninteresting-conditions*.

`*warnings-file-type*` [Variable]
Pathname type for warnings files, or `nil` if disabled

12 UIOP/LAUNCH-PROGRAM

`uiop/launch-program` semi-portably launches a program as an asynchronous external subprocess. Available functionality may depend on the underlying implementation.

`process-info` [Class]

Class precedence list: `process-info, standard-object, t`

This class should be treated as opaque by programmers, except for the exported `process-info-*` functions. It should never be directly instantiated by `make-instance`. Primarily, it is being made available to enable type-checking.

`close-streams` *process-info* [Function]

Close any stream that the process might own. Needs to be run whenever streams were requested by passing :stream to :input, :output, or :error-output.

`easy-sh-character-p` *x* [Function]

Is `x` an "easy" character that does not require quoting by the shell?

`escape-command` *command &optional s escaper* [Function]

Given a `command` as a list of tokens, return a string of the spaced, escaped tokens, using `escaper` to escape.

`escape-sh-command` *command &optional s* [Function]

Escape a list of command-line arguments into a string suitable for parsing by /bin/sh in POSIX

`escape-sh-token` *token &optional s* [Function]

Escape a string `token` within double-quotes if needed for use within a POSIX Bourne shell, outputing to `s`.

`escape-shell-command` *command &optional stream* [Function]

Escape a command for the current operating system's shell

`escape-shell-token` *token &optional s* [Function]

Escape a token for the current operating system shell

`escape-token` *token &key stream quote good-chars bad-chars escaper* [Function]

Call the `escaper` function on `token` string if it needs escaping as per `requires-escaping-p` using `good-chars` and `bad-chars`, otherwise output `token`, using `stream` as output (or returning result as a string if `nil`)

`escape-windows-command` *command &optional s* [Function]

Escape a list of command-line arguments into a string suitable for parsing by CommandLineToArgv in `ms` Windows

`escape-windows-token` *token &optional s* [Function]

Escape a string `token` within double-quotes if needed for use within a `ms` Windows command-line, outputing to `s`.

`launch-program` *command &rest keys &key input if-input-does-not-exist output if-output-exists error-output if-error-output-exists element-type external-format directory &allow-other-keys* [Function]

Launch program specified by `command`, either a list of strings specifying a program and list of arguments, or a string specifying a shell command (/bin/sh on Unix, `cmd.exe` on Windows) _asynchronously_.

If `output` is a pathname, a string designating a pathname, or `nil` (the default) designating the null device, the file at that path is used as output. If it's `:interactive`, output is inherited from the current process; beware that this may be different from your `*standard-output*`, and under `slime` will be on your *inferior-lisp* buffer. If it's `t`, output goes to your current `*standard-output*` stream. If it's `:stream`, a new stream will be made available that can be accessed via `process-info-output` and read from. Otherwise, `output` should be a value that the underlying lisp implementation knows how to handle.

`if-output-exists`, which is only meaningful if `output` is a string or a pathname, can take the values `:error`, `:append`, and `:supersede` (the default). The meaning of these values and their effect on the case where `output` does not exist, is analogous to the `if-exists` parameter to `open` with `:direction :output`.

`error-output` is similar to `output`. `t` designates the `*error-output*`, `:output` means redirecting the error output to the output stream, and `:stream` causes a stream to be made available via `process-info-error-output`.

`if-error-output-exists` is similar to `if-output-exist`, except that it affects `error-output` rather than `output`.

`input` is similar to `output`, except that `t` designates the `*standard-input*` and a stream requested through the `:stream` keyword would be available through `process-info-input`.

`if-input-does-not-exist`, which is only meaningful if `input` is a string or a pathname, can take the values `:create` and `:error` (the default). The meaning of these values is analogous to the `if-does-not-exist` parameter to `open` with `:direction :input`.

`element-type` and `external-format` are passed on to your Lisp implementation, when applicable, for creation of the output stream.

`launch-program` returns a `process-info` object.

`launch-program` currently does not smooth over all the differences between implementations. Of particular note is when streams are provided for `output` or `error-output`. Some implementations don't support this at all, some support only certain subclasses of streams, and some support any arbitrary stream. Additionally, the implementations that support streams may have differing behavior on how those streams are filled with data. If data is not periodically read from the child process and sent to the stream, the child could block because its output buffers are full.

`process-alive-p` *process-info* [Function]

Check if a process has yet to exit.

`terminate-process` *process-info &key urgent* [Function]

Cause the process to exit. To that end, the process may or may not be sent a signal, which it will find harder (or even impossible) to ignore if `urgent` is `t`. On some platforms, it may also be subject to race conditions.

`wait-process` *process-info* [Function]

Wait for the process to terminate, if it is still running. Otherwise, return immediately. An exit code (a number) will be returned, with `0` indicating success, and anything else indicating failure. If the process exits after receiving a signal, the exit code will be the sum of `128` and the (positive) numeric signal code. A second value may be returned in this case: the numeric signal code itself. Any asynchronously spawned process requires this function to be run before it is garbage-collected in order to free up resources that might otherwise be irrevocably lost.

13 UIOP/RUN-PROGRAM

`uiop/run-program` fully portably runs a program as a synchronous external subprocess.

`run-program` *command &rest keys &key ignore-error-status force-shell input if-input-does-not-exist output if-output-exists error-output if-error-output-exists element-type external-format &allow-other-keys* [Function]

Run program specified by `command`, either a list of strings specifying a program and list of arguments, or a string specifying a shell command (/bin/sh on Unix, `cmd.exe` on Windows); _synchronously_ process its output as specified and return the processing results when the program and its output processing are complete.

Always call a shell (rather than directly execute the command when possible) if `force-shell` is specified. Similarly, never call a shell if `force-shell` is specified to be `nil`.

Signal a continuable `subprocess-error` if the process wasn't successful (exit-code `0`), unless `ignore-error-status` is specified.

If `output` is a pathname, a string designating a pathname, or `nil` (the default) designating the null device, the file at that path is used as output. If it's `:interactive`, output is inherited from the current process; beware that this may be different from your `*standard-output*`, and under `slime` will be on your *inferior-lisp* buffer. If it's `t`, output goes to your current `*standard-output*` stream. Otherwise, `output` should be a value that is a suitable first argument to `slurp-input-stream` (qv.), or a list of such a value and keyword arguments. In this case, `run-program` will create a temporary stream for the program output; the program output, in that stream, will be processed by a call to `slurp-input-stream`, using `output` as the first argument (or the first element of `output`, and the rest as keywords). The primary value resulting from that call (or `nil` if no call was needed) will be the first value returned by `run-program`. `e`.g., using `:output :string` will have it return the entire output stream as a string. And using `:output '(:string :stripped t)` will have it return the same string stripped of any ending newline.

`if-output-exists`, which is only meaningful if `output` is a string or a pathname, can take the values `:error`, `:append`, and `:supersede` (the default). The meaning of these values and their effect on the case where `output` does not exist, is analogous to the `if-exists` parameter to `open` with `:direction :output`.

`error-output` is similar to `output`, except that the resulting value is returned as the second value of `run-program`. `t` designates the `*error-output*`. Also `:output` means redirecting the error output to the output stream, in which case `nil` is returned.

`if-error-output-exists` is similar to `if-output-exist`, except that it affects `error-output` rather than `output`.

`input` is similar to `output`, except that `vomit-output-stream` is used, no value is returned, and `t` designates the `*standard-input*`.

`if-input-does-not-exist`, which is only meaningful if `input` is a string or a pathname, can take the values `:create` and `:error` (the default). The meaning of these values is analogous to the `if-does-not-exist` parameter to `open` with `:direction :input`.

`element-type` and `external-format` are passed on to your Lisp implementation, when applicable, for creation of the output stream.

One and only one of the stream slurping or vomiting may or may not happen in parallel in parallel with the subprocess, depending on options and implementation, and with priority being given to output processing. Other streams are completely produced or consumed before or after the subprocess is spawned, using temporary files.

`run-program` returns 3 values: `0-` the result of the `output` slurping if any, or `nil` `1-` the result of the `error-output` slurping if any, or `nil` `2-` either `0` if the subprocess exited with success status, or an indication of failure via the `exit-code` of the process

`slurp-input-stream` *processor input-stream &key linewise prefix element-type buffer-size external-format if-exists if-does-not-exist at count stripped &allow-other-keys* [Generic Function]

`slurp-input-stream` is a generic function with two positional arguments `processor` and `input-stream` and additional keyword arguments, that consumes (slurps) the contents of the `input-stream` and processes them according to a method specified by `processor`.

Built-in methods include the following:

- if `processor` is a function, it is called with the `input-stream` as its argument
- if `processor` is a list, its first element should be a function. It will be applied to a cons of the `input-stream` and the rest of the list. That is (x . y) will be treated as (`apply` x <stream> y)
- if `processor` is an output-stream, the contents of `input-stream` is copied to the output-stream, per copy-stream-to-stream, with appropriate keyword arguments.
- if `processor` is the symbol `cl:string` or the keyword `:string`, then the contents of `input-stream` are returned as a string, as per `slurp-stream-string`.
- if `processor` is the keyword `:lines` then the `input-stream` will be handled by `slurp-stream-lines`.
- if `processor` is the keyword `:line` then the `input-stream` will be handled by `slurp-stream-line`.
- if `processor` is the keyword `:forms` then the `input-stream` will be handled by `slurp-stream-forms`.
- if `processor` is the keyword `:form` then the `input-stream` will be handled by `slurp-stream-form`.
- if `processor` is `t`, it is treated the same as *standard-output*. If it is `nil`, `nil` is returned.

Programmers are encouraged to define their own methods for this generic function.

`vomit-output-stream` *processor output-stream &key linewise prefix element-type buffer-size external-format if-exists if-does-not-exist fresh-line terpri &allow-other-keys* [Generic Function]

`vomit-output-stream` is a generic function with two positional arguments `processor` and `output-stream` and additional keyword arguments, that produces (vomits) some content onto the `output-stream`, according to a method specified by `processor`.

Built-in methods include the following:

- if `processor` is a function, it is called with the `output-stream` as its argument
- if `processor` is a list, its first element should be a function. It will be applied to a cons of the `output-stream` and the rest of the list. That is (x . y) will be treated as (`apply` x `<stream>` y)
- if `processor` is an input-stream, its contents will be copied the `output-stream`, per copy-stream-to-stream, with appropriate keyword arguments.
- if `processor` is a string, its contents will be printed to the `output-stream`.
- if `processor` is `t`, it is treated the same as *standard-input*. If it is `nil`, nothing is done.

Programmers are encouraged to define their own methods for this generic function.

14 UIOP/CONFIGURATION

`clear-configuration` [Function]
Call the functions in `*clear-configuration-hook*`

`configuration-inheritance-directive-p` *x* [Function]
Is `x` a configuration inheritance directive?

`filter-pathname-set` *dirs* [Function]
Parse strings as unix namestrings and remove duplicates and non absolute-pathnames in a list.

`find-preferred-file` *files &key direction* [Function]
Find first file in the list of `files` that exists (for direction :input or :probe) or just the first one (for direction :output or :io). Note that when we say "file" here, the files in question may be directories.

`get-folder-path` *folder* [Function]
Semi-portable implementation of a subset of LispWorks' sys:get-folder-path, this function tries to locate the Windows `folder` for one of `:local-appdata`, `:appdata` or `:common-appdata`. Returns `nil` when the folder is not defined (e.g., not on Windows).

`in-first-directory` *dirs x &key direction* [Function]
Finds the first appropriate file named `x` in the list of `dirs` for I/O in `direction` (which may be `:input`, `:output`, `:io`, or `:probe`). If direction is `:input` or `:probe`, will return the first extant file named `x` in one of the `dirs`. If direction is `:output` or `:io`, will simply return the file named `x` in the first element of `dirs` that exists. `deprecated`.

`in-system-configuration-directory` *x &key direction* [Function]
Return the pathname for the file named `x` under the system configuration directory for common-lisp. `deprecated`.

`in-user-configuration-directory` *x &key direction* [Function]
Return the file named `x` in the user configuration directory for common-lisp. `deprecated`.

`location-designator-p` *x* [Function]
Is `x` a designator for a location?

`location-function-p` *x* [Function]
Is `x` the specification of a location function?

`register-clear-configuration-hook` *hook-function &optional call-now-p* [Function]
Register a function to be called when clearing configuration

`report-invalid-form` *reporter &rest args* [Function]
Report an invalid form according to `reporter` and various `args`

`resolve-absolute-location` *x &key ensure-directory wilden* [Function]
Given a designator `x` for an absolute location, resolve it to a pathname

`resolve-location` *x &key ensure-directory wilden directory* [Function]
Resolve location designator `x` into a `pathname`

`resolve-relative-location` *x &key ensure-directory wilden* [Function]
Given a designator `x` for an relative location, resolve it to a pathname.

`system-config-pathnames` *&rest more* [Function]
Return a list of directories where are stored the system's default user configuration information. `more` may contain specifications for a subpath relative to these directories: a subpathname specification and keyword arguments as per `resolve-location` (see also "Configuration DSL") in the ASDF manual.

`system-configuration-directories` [Function]
Return the list of system configuration directories for common-lisp. `deprecated`. Use `uiop:system-config-pathnames` (with argument "common-lisp"), instead.

`uiop-directory` [Function]
Try to locate the `uiop` source directory at runtime

`upgrade-configuration` [Function]
If a previous version of ASDF failed to read some configuration, try again now.

`user-configuration-directories` [Function]
Return the current user's list of user configuration directories for configuring common-lisp. `deprecated`. Use `uiop:xdg-config-pathnames` instead.

`validate-configuration-directory` *directory tag validator &key invalid-form-reporter* [Function]
Map the `validator` across the .conf files in `directory`, the `tag` will be applied to the results to yield a configuration form. Current values of `tag` include :source-registry and :output-translations.

`validate-configuration-file` *file validator &key description* [Function]
Validate a configuration `file`. The configuration file should have only one s-expression in it, which will be checked with the `validator` `form`. `description` argument used for error reporting.

`validate-configuration-form` *form tag directive-validator &key location invalid-form-reporter* [Function]
Validate a configuration `form`. By default it will raise an error if the `form` is not valid. Otherwise it will return the validated form. Arguments control the behavior: The configuration `form` should be of the form (`tag` . <rest>) Each element of <rest> will be checked by first seeing if it's a configuration inheritance directive (see `configuration-inheritance-directive-p`) then invoking `directive-validator` on it. In the event of an invalid form, `invalid-form-reporter` will be used to control reporting (see `report-invalid-form`) with `location` providing information about where the configuration form appeared.

`xdg-cache-home` *&rest more* [Function]
The base directory relative to which user specific non-essential data files should be stored. Returns an absolute directory pathname. `more` may contain specifications for a subpath relative to this directory: a subpathname specification and keyword arguments as per `resolve-location` (see also "Configuration DSL") in the ASDF manual.

`xdg-config-dirs` *&rest more* [Function]
The preference-ordered set of additional base paths to search for configuration files. Returns a list of absolute directory pathnames. `more` may contain specifications for a subpath relative to these directories: subpathname specification and keyword arguments as per `resolve-location` (see also "Configuration DSL") in the ASDF manual.

`xdg-config-home` *&rest more* [Function]
Returns a pathname for the directory containing user-specific configuration files. `more` may contain specifications for a subpath relative to this directory: a subpathname specification and keyword arguments as per `resolve-location` (see also "Configuration DSL") in the ASDF manual.

`xdg-config-pathnames` *&rest more* [Function]
Return a list of pathnames for application configuration. `more` may contain specifications for a subpath relative to these directories: a subpathname specification and keyword arguments as per `resolve-location` (see also "Configuration DSL") in the ASDF manual.

`xdg-data-dirs` *&rest more* [Function]
The preference-ordered set of additional paths to search for data files. Returns a list of absolute directory pathnames. `more` may contain specifications for a subpath relative to these directories: a subpathname specification and keyword arguments as per `resolve-location` (see also "Configuration DSL") in the ASDF manual.

`xdg-data-home` *&rest more* [Function]
Returns an absolute pathname for the directory containing user-specific data files. `more` may contain specifications for a subpath relative to this directory: a subpathname specification and keyword arguments as per `resolve-location` (see also "Configuration DSL") in the ASDF manual.

`xdg-data-pathnames` *&rest more* [Function]
Return a list of absolute pathnames for application data directories. With `app`, returns directory for data for that application, without `app`, returns the set of directories for storing all application configurations. `more` may contain specifications for a subpath relative to these directories: a subpathname specification and keyword arguments as per `resolve-location` (see also "Configuration DSL") in the ASDF manual.

`xdg-runtime-dir` *&rest more* [Function]
Pathname for user-specific non-essential runtime files and other file objects, such as sockets, named pipes, etc. Returns an absolute directory pathname. `more` may

contain specifications for a subpath relative to this directory: a subpathname specification and keyword arguments as per `resolve-location` (see also "Configuration DSL") in the ASDF manual.

`*here-directory*` [Variable]
This special variable is bound to the currect directory during calls to `process-source-registry` in order that we be able to interpret the :here directive.

`*ignored-configuration-form*` [Variable]
Have configuration forms been ignored while parsing the configuration?

`*user-cache*` [Variable]
A specification as per `resolve-location` of where the user keeps his FASL cache

15 UIOP/BACKWARD-DRIVER

`uiop/backward-driver` provides backward-compatibility with earlier incarnations of this library.

`coerce-pathname` *name &key type defaults* [Function]
: `deprecated`. Please use `uiop:parse-unix-namestring` instead.

`in-first-directory` *dirs x &key direction* [Function]
: Finds the first appropriate file named `x` in the list of `dirs` for I/O in `direction` (which may be `:input`, `:output`, `:io`, or `:probe`). If direction is `:input` or `:probe`, will return the first extant file named `x` in one of the `dirs`. If direction is `:output` or `:io`, will simply return the file named `x` in the first element of `dirs` that exists. `deprecated`.

`in-system-configuration-directory` *x &key direction* [Function]
: Return the pathname for the file named `x` under the system configuration directory for common-lisp. `deprecated`.

`in-user-configuration-directory` *x &key direction* [Function]
: Return the file named `x` in the user configuration directory for common-lisp. `deprecated`.

`system-configuration-directories` [Function]
: Return the list of system configuration directories for common-lisp. `deprecated`. Use `uiop:system-config-pathnames` (with argument `"common-lisp"`), instead.

`user-configuration-directories` [Function]
: Return the current user's list of user configuration directories for configuring common-lisp. `deprecated`. Use `uiop:xdg-config-pathnames` instead.

`version-compatible-p` *provided-version required-version* [Function]
: Is the provided version a compatible substitution for the required-version? If major versions differ, it's not compatible. If they are equal, then any later version is compatible, with later being determined by a lexicographical comparison of minor numbers. `deprecated`.

16 UIOP/DRIVER

`uiop/driver` doesn't export any new symbols. It just exists to reexport all the utilities in a single package `uiop`.