Version 15.17R, X3J13/94-101R.
Fri 12-Aug-1994 6:35pm EDT

Programming Language—Common Lisp

19. Filenames

19.1 Overview of Filenames

There are many kinds of *file systems*, varying widely both in their superficial syntactic details, and in their underlying power and structure. The facilities provided by Common Lisp for referring to and manipulating *files* has been chosen to be compatible with many kinds of *file systems*, while at the same time minimizing the program-visible differences between kinds of *file systems*.

Since *file systems* vary in their conventions for naming *files*, there are two distinct ways to represent *filenames*: as *namestrings* and as *pathnames*.

19.1.1 Namestrings as Filenames

A ***namestring*** is a *string* that represents a *filename*.

In general, the syntax of *namestrings* involves the use of *implementation-defined* conventions, usually those customary for the *file system* in which the named *file* resides. The only exception is the syntax of a *logical pathname namestring*, which is defined in this specification; see Section 19.3.1 (Syntax of Logical Pathname Namestrings).

A *conforming program* must never unconditionally use a *literal namestring* other than a *logical pathname namestring* because Common Lisp does not define any *namestring* syntax other than that for *logical pathnames* that would be guaranteed to be portable. However, a *conforming program* can, if it is careful, successfully manipulate user-supplied data which contains or refers to non-portable *namestrings*.

A *namestring* can be *coerced* to a *pathname* by the *functions* **pathname** or **parse-namestring**.

19.1.2 Pathnames as Filenames

Pathnames are structured *objects* that can represent, in an *implementation-independent* way, the *filenames* that are used natively by an underlying *file system*.

In addition, *pathnames* can also represent certain partially composed *filenames* for which an underlying *file system* might not have a specific *namestring* representation.

A *pathname* need not correspond to any file that actually exists, and more than one *pathname* can refer to the same file. For example, the *pathname* with a version of `:newest` might refer to the same file as a *pathname* with the same components except a certain number as the version. Indeed, a *pathname* with version `:newest` might refer to different files as time passes, because the meaning of such a *pathname* depends on the state of the file system.

Some *file systems* naturally use a structural model for their *filenames*, while others do not. Within the Common Lisp *pathname* model, all *filenames* are seen as having a particular structure, even if that structure is not reflected in the underlying *file system*. The nature of the mapping between structure imposed by *pathnames* and the structure, if any, that is used by the underlying *file system* is *implementation-defined*.

Every *pathname* has six components: a host, a device, a directory, a name, a type, and a version. By naming *files* with *pathnames*, Common Lisp programs can work in essentially the same way even in *file systems* that seem superficially quite different. For a detailed description of these components, see Section 19.2.1 (Pathname Components).

The mapping of the *pathname* components into the concepts peculiar to each *file system* is *implementation-defined*. There exist conceivable *pathnames* for which there is no mapping to a syntactically valid *filename* in a particular *implementation*. An *implementation* may use various strategies in an attempt to find a mapping; for example, an *implementation* may quietly truncate *filenames* that exceed length limitations imposed by the underlying *file system*, or ignore certain *pathname* components for which the *file system* provides no support. If such a mapping cannot be found, an error of *type* **file-error** is signaled.

The time at which this mapping and associated error signaling occurs is *implementation-dependent*. Specifically, it may occur at the time the *pathname* is constructed, when coercing a *pathname* to a *namestring*, or when an attempt is made to *open* or otherwise access the *file* designated by the *pathname*.

Figure 19–1 lists some *defined names* that are applicable to *pathnames*.

default-pathname-defaults	**namestring**	**pathname-name**
directory-namestring	**open**	**pathname-type**
enough-namestring	**parse-namestring**	**pathname-version**
file-namestring	**pathname**	**pathnamep**
file-string-length	**pathname-device**	**translate-pathname**
host-namestring	**pathname-directory**	**truename**
make-pathname	**pathname-host**	**user-homedir-pathname**
merge-pathnames	**pathname-match-p**	**wild-pathname-p**

Figure 19–1. Pathname Operations

19.1.3 Parsing Namestrings Into Pathnames

Parsing is the operation used to convert a *namestring* into a *pathname*. Except in the case of parsing *logical pathname namestrings*, this operation is *implementation-dependent*, because the format of *namestrings* is *implementation-dependent*.

A *conforming implementation* is free to accommodate other *file system* features in its *pathname* representation and provides a parser that can process such specifications in *namestrings*. *Conforming programs* must not depend on any such features, since those features will not be portable.

19.2 Pathnames

19.2.1 Pathname Components

A *pathname* has six components: a host, a device, a directory, a name, a type, and a version.

19.2.1.1 The Pathname Host Component

The name of the file system on which the file resides, or the name of a *logical host*.

19.2.1.2 The Pathname Device Component

Corresponds to the "device" or "file structure" concept in many host file systems: the name of a logical or physical device containing files.

19.2.1.3 The Pathname Directory Component

Corresponds to the "directory" concept in many host file systems: the name of a group of related files.

19.2.1.4 The Pathname Name Component

The "name" part of a group of *files* that can be thought of as conceptually related.

19.2.1.5 The Pathname Type Component

Corresponds to the "filetype" or "extension" concept in many host file systems. This says what kind of file this is. This component is always a *string*, **nil**, `:wild`, or `:unspecific`.

19.2.1.6 The Pathname Version Component

Corresponds to the "version number" concept in many host file systems.

The version is either a positive *integer* or a *symbol* from the following list: **nil**, `:wild`, `:unspecific`, or `:newest` (refers to the largest version number that already exists in the file system when reading a file, or to a version number greater than any already existing in the file system when writing a new file). Implementations can define other special version *symbols*.

19.2.2 Interpreting Pathname Component Values

19.2.2.1 Strings in Component Values

19.2.2.1.1 Special Characters in Pathname Components

Strings in *pathname* component values never contain special *characters* that represent separation between *pathname* fields, such as *slash* in Unix *filenames*. Whether separator *characters* are permitted as part of a *string* in a *pathname* component is *implementation-defined*; however, if the *implementation* does permit it, it must arrange to properly "quote" the character for the *file system* when constructing a *namestring*. For example,

```
 ;; In a TOPS-20 implementation, which uses ^V to quote
 (NAMESTRING (MAKE-PATHNAME :HOST "OZ" :NAME "<TEST>"))
→ #P"OZ:PS:^V<TEST^V>"
```

$\overset{not}{\rightarrow}$ `#P"OZ:PS:<TEST>"`

19.2.2.1.2 Case in Pathname Components

Namestrings always use local file system *case* conventions, but Common Lisp *functions* that manipulate *pathname* components allow the caller to select either of two conventions for representing *case* in component values by supplying a value for the `:case` keyword argument. Figure 19–2 lists the functions relating to *pathnames* that permit a `:case` argument:

make-pathname	**pathname-directory**	**pathname-name**
pathname-device	**pathname-host**	**pathname-type**

Figure 19–2. Pathname functions using a :CASE argument

19.2.2.1.2.1 Local Case in Pathname Components

For the functions in Figure 19–2, a value of `:local` for the `:case` argument (the default for these functions) indicates that the functions should receive and yield *strings* in component values as if they were already represented according to the host *file system*'s convention for *case*.

If the *file system* supports both *cases*, *strings* given or received as *pathname* component values under this protocol are to be used exactly as written. If the file system only supports one *case*, the *strings* will be translated to that *case*.

19.2.2.1.2.2 Common Case in Pathname Components

For the functions in Figure 19–2, a value of `:common` for the `:case` argument that these *functions* should receive and yield *strings* in component values according to the following conventions:

- All *uppercase* means to use a file system's customary *case*.
- All *lowercase* means to use the opposite of the customary *case*.
- Mixed *case* represents itself.

Note that these conventions have been chosen in such a way that translation from `:local` to `:common` and back to `:local` is information-preserving.

19.2.2.2 Special Pathname Component Values

19.2.2.2.1 NIL as a Component Value

As a *pathname* component value, **nil**represents that the component is "unfilled"; see Section 19.2.3 (Merging Pathnames).

The value of any *pathname* component can be **nil**.

When constructing a *pathname*, **nil** in the host component might mean a default host rather than an actual **nil** in some *implementations*.

19.2.2.2.2 :WILD as a Component Value

If `:wild` is the value of a *pathname* component, that component is considered to be a wildcard, which matches anything.

A *conforming program* must be prepared to encounter a value of `:wild` as the value of any *pathname* component, or as an *element* of a *list* that is the value of the directory component.

When constructing a *pathname*, a *conforming program* may use `:wild` as the value of any or all of the directory, name, type, or version component, but must not use `:wild` as the value of the host, or device component.

If `:wild` is used as the value of the directory component in the construction of a *pathname*, the effect is equivalent to specifying the list `(:absolute :wild-inferiors)`, or the same as `(:absolute :wild)` in a *file system* that does not support `:wild-inferiors`.

19.2.2.2.3 :UNSPECIFIC as a Component Value

If `:unspecific` is the value of a *pathname* component, the component is considered to be "absent" or to "have no meaning" in the *filename* being represented by the *pathname*.

Whether a value of `:unspecific` is permitted for any component on any given *file system* accessible to the *implementation* is *implementation-defined*. A *conforming program* must never unconditionally use a `:unspecific` as the value of a *pathname* component because such a value is not guaranteed to be permissible in all implementations. However, a *conforming program* can, if it is careful, successfully manipulate user-supplied data which contains or refers to non-portable *pathname* components. And certainly a *conforming program* should be prepared for the possibility that any components of a *pathname* could be `:unspecific`.

When *reading*$_1$ the value of any *pathname* component, *conforming programs* should be prepared for the value to be `:unspecific`.

When *writing*$_1$ the value of any *pathname* component, the consequences are undefined if `:unspecific` is given for a *pathname* in a *file system* for which it does not make sense.

19.2.2.2.3.1 Relation between component values NIL and :UNSPECIFIC

If a *pathname* is converted to a *namestring*, the *symbols* **nil** and `:unspecific` cause the field to be treated as if it were empty. That is, both **nil** and `:unspecific` cause the component not to appear in the *namestring*.

However, when merging a *pathname* with a set of defaults, only a **nil** value for a component will be replaced with the default for that component, while a value of `:unspecific` will be left alone as if the field were "filled"; see the *function* **merge-pathnames** and Section 19.2.3 (Merging Pathnames).

19.2.2.3 Restrictions on Wildcard Pathnames

Wildcard *pathnames* can be used with **directory** but not with **open**, and return true from **wild-pathname-p**. When examining wildcard components of a wildcard *pathname*, conforming programs must be prepared to encounter any of the following additional values in any component or any element of a *list* that is the directory component:

- The *symbol* `:wild`, which matches anything.
- A *string* containing *implementation-dependent* special wildcard *characters*.
- Any *object*, representing an *implementation-dependent* wildcard pattern.

19.2.2.4 Restrictions on Examining Pathname Components

The space of possible *objects* that a *conforming program* must be prepared to *read*$_1$ as the value of a *pathname* component is substantially larger than the space of possible *objects* that a *conforming program* is permitted to *write*$_1$ into such a component.

While the values discussed in the subsections of this section, in Section 19.2.2.2 (Special Pathname Component Values), and in Section 19.2.2.3 (Restrictions on Wildcard Pathnames) apply to values that might be seen when reading the component values, substantially more restrictive rules apply to constructing pathnames; see Section 19.2.2.5 (Restrictions on Constructing Pathnames).

When examining *pathname* components, *conforming programs* should be aware of the following restrictions.

19.2.2.4.1 Restrictions on Examining a Pathname Host Component

It is *implementation-dependent* what *object* is used to represent the host.

19.2.2.4.2 Restrictions on Examining a Pathname Device Component

The device might be a *string*, `:wild`, `:unspecific`, or **nil**.

Note that `:wild` might result from an attempt to *read*$_1$ the *pathname* component, even though portable programs are restricted from *writing*$_1$ such a component value; see Section 19.2.2.3 (Restrictions on Wildcard Pathnames) and Section 19.2.2.5 (Restrictions on Constructing Pathnames).

19.2.2.4.3 Restrictions on Examining a Pathname Directory Component

The directory might be a *string*, `:wild`, `:unspecific`, or **nil**.

The directory can be a *list* of *strings* and *symbols*. The *car* of the *list* is one of the symbols `:absolute` or `:relative`, meaning:

`:absolute`

A *list* whose *car* is the symbol `:absolute` represents a directory path starting from the root directory. The list `(:absolute)` represents the root directory. The list `(:absolute "foo" "bar" "baz")` represents the directory called `"/foo/bar/baz"` in Unix (except possibly for *case*).

`:relative`

A *list* whose *car* is the symbol `:relative` represents a directory path starting from a default directory. The list `(:relative)` has the same meaning as **nil** and hence is not used. The list `(:relative "foo" "bar")` represents the directory named `"bar"` in the directory named `"foo"` in the default directory.

Each remaining element of the *list* is a *string* or a *symbol*.

Each *string* names a single level of directory structure. The *strings* should contain only the directory names themselves—no punctuation characters.

In place of a *string*, at any point in the *list*, *symbols* can occur to indicate special file notations. Figure 19–3 lists the *symbols* that have standard meanings. Implementations are permitted to add additional *objects* of any *type* that is disjoint from **string** if necessary to represent features of their file systems that cannot be represented with the standard *strings* and *symbols*.

Supplying any non-*string*, including any of the *symbols* listed below, to a file system for which it does not make sense signals an error of *type* **file-error**. For example, Unix does not support `:wild-inferiors` in most implementations.

Symbol	**Meaning**
`:wild`	Wildcard match of one level of directory structure
`:wild-inferiors`	Wildcard match of any number of directory levels
`:up`	Go upward in directory structure (semantic)
`:back`	Go upward in directory structure (syntactic)

Figure 19–3. Special Markers In Directory Component

The following notes apply to the previous figure:

Invalid Combinations

Using `:absolute` or `:wild-inferiors` immediately followed by `:up` or `:back` signals an error of *type* **file-error**.

Syntactic vs Semantic

“Syntactic” means that the action of `:back` depends only on the *pathname* and not on the contents of the file system.

“Semantic” means that the action of `:up` depends on the contents of the file system; to resolve a *pathname* containing `:up` to a *pathname* whose directory component contains only `:absolute` and *strings* requires probing the file system.

`:up` differs from `:back` only in file systems that support multiple names for directories, perhaps via symbolic links. For example, suppose that there is a directory `(:absolute "X" "Y" "Z")` linked to `(:absolute "A" "B" "C")` and there also exist directories `(:absolute "A" "B" "Q")` and `(:absolute "X" "Y" "Q")`. Then `(:absolute "X" "Y" "Z" :up "Q")` designates `(:absolute "A" "B" "Q")` while `(:absolute "X" "Y" "Z" :back "Q")` designates `(:absolute "X" "Y" "Q")`

19.2.2.4.3.1 Directory Components in Non-Hierarchical File Systems

In non-hierarchical *file systems*, the only valid *list* values for the directory component of a *pathname* are `(:absolute` *string*`)` and `(:absolute :wild)`. `:relative` directories and the keywords `:wild-inferiors`, `:up`, and `:back` are not used in non-hierarchical *file systems*.

19.2.2.4.4 Restrictions on Examining a Pathname Name Component

The name might be a *string*, `:wild`, `:unspecific`, or **nil**.

19.2.2.4.5 Restrictions on Examining a Pathname Type Component

The type might be a *string*, `:wild`, `:unspecific`, or **nil**.

19.2.2.4.6 Restrictions on Examining a Pathname Version Component

The version can be any *symbol* or any *integer*.

The symbol `:newest` refers to the largest version number that already exists in the *file system* when reading, overwriting, appending, superseding, or directory listing an existing *file*. The symbol `:newest` refers to the smallest version number greater than any existing version number when creating a new file.

The symbols **nil**, `:unspecific`, and `:wild` have special meanings and restrictions; see Section 19.2.2.2 (Special Pathname Component Values) and Section 19.2.2.5 (Restrictions on Constructing Pathnames).

Other *symbols* and *integers* have *implementation-defined* meaning.

19.2.2.4.7 Notes about the Pathname Version Component

It is suggested, but not required, that implementations do the following:

- Use positive *integers* starting at 1 as version numbers.
- Recognize the symbol `:oldest` to designate the smallest existing version number.
- Use *keywords* for other special versions.

19.2.2.5 Restrictions on Constructing Pathnames

When constructing a *pathname* from components, conforming programs must follow these rules:

- Any component can be **nil**. **nil** in the host might mean a default host rather than an actual **nil** in some implementations.
- The host, device, directory, name, and type can be *strings*. There are *implementation-dependent* limits on the number and type of *characters* in these *strings*.
- The directory can be a *list* of *strings* and *symbols*. There are *implementation-dependent* limits on the *list*'s length and contents.
- The version can be `:newest`.
- Any component can be taken from the corresponding component of another *pathname*. When the two *pathnames* are for different file systems (in implementations that support multiple file systems), an appropriate translation occurs. If no meaningful translation is possible, an error is signaled. The definitions of "appropriate" and "meaningful" are *implementation-dependent*.
- An implementation might support other values for some components, but a portable program cannot use those values. A conforming program can use *implementation-dependent* values but this can make it non-portable; for example, it might work only with Unix file systems.

19.2.3 Merging Pathnames

Merging takes a *pathname* with unfilled components and supplies values for those components from a source of defaults.

If a component's value is **nil**, that component is considered to be unfilled. If a component's value is any *non-nil object*, including `:unspecific`, that component is considered to be filled.

Except as explicitly specified otherwise, for functions that manipulate or inquire about *files* in the *file system*, the pathname argument to such a function is merged with ***default-pathname-defaults*** before accessing the *file system* (as if by **merge-pathnames**).

19.2.3.1 Examples of Merging Pathnames

Although the following examples are possible to execute only in *implementations* which permit `:unspecific` in the indicated position andwhich permit four-letter type components, they serve to illustrate the basic concept of *pathname* merging.

```
 (pathname-type
   (merge-pathnames (make-pathname :type "LISP")
                    (make-pathname :type "TEXT")))
→ "LISP"


 (pathname-type
   (merge-pathnames (make-pathname :type nil)
                    (make-pathname :type "LISP")))
→ "LISP"


 (pathname-type
   (merge-pathnames (make-pathname :type :unspecific)
                    (make-pathname :type "LISP")))
→ :UNSPECIFIC
```

19.3 Logical Pathnames

19.3.1 Syntax of Logical Pathname Namestrings

The syntax of a *logical pathname namestring* is as follows. (Note that unlike many notational descriptions in this document, this is a syntactic description of character sequences, not a structural description of *objects*.)

logical-pathname::=[↓*host* *host-marker*]
[↓*relative-directory-marker*] {↓*directory* *directory-marker*}*
[↓*name*] [*type-marker* ↓*type* [*version-marker* ↓*version*]]

host::=↓*word*

directory::=↓*word* | ↓*wildcard-word* | ↓*wild-inferiors-word*

name::=↓*word* | ↓*wildcard-word*

type::=↓*word* | ↓*wildcard-word*

version::=↓*pos-int* | *newest-word* | *wildcard-version*

host-marker—a *colon*.

relative-directory-marker—a *semicolon*.

directory-marker—a *semicolon*.

type-marker—a *dot*.

version-marker—a *dot*.

wild-inferiors-word—The two character sequence "`**`" (two *asterisks*).

newest-word—The six character sequence "`newest`" or the six character sequence "`NEWEST`".

wildcard-version—an *asterisk*.

wildcard-word—one or more *asterisks*, uppercase letters, digits, and hyphens, including at least one *asterisk*, with no two *asterisks* adjacent.

word—one or more uppercase letters, digits, and hyphens.

pos-int—a positive *integer*.

19.3.1.1 Additional Information about Parsing Logical Pathname Namestrings

19.3.1.1.1 The Host part of a Logical Pathname Namestring

The *host* must have been defined as a *logical pathname* host; this can be done by using **setf** of **logical-pathname-translations**.

The *logical pathname* host name `"SYS"` is reserved for the implementation. The existence and meaning of `SYS:` *logical pathnames* is *implementation-defined*.

19.3.1.1.2 The Device part of a Logical Pathname Namestring

There is no syntax for a *logical pathname* device since the device component of a *logical pathname* is always `:unspecific`; see Section 19.3.2.1 (Unspecific Components of a Logical Pathname).

19.3.1.1.3 The Directory part of a Logical Pathname Namestring

If a *relative-directory-marker* precedes the *directories*, the directory component parsed is as *relative*; otherwise, the directory component is parsed as *absolute*.

If a *wild-inferiors-marker* is specified, it parses into `:wild-inferiors`.

19.3.1.1.4 The Type part of a Logical Pathname Namestring

The *type* of a *logical pathname* for a *source file* is `"LISP"`. This should be translated into whatever type is appropriate in a physical pathname.

19.3.1.1.5 The Version part of a Logical Pathname Namestring

Some *file systems* do not have *versions*. *Logical pathname* translation to such a *file system* ignores the *version*. This implies that a program cannot rely on being able to store more than one version of a file named by a *logical pathname*.

If a *wildcard-version* is specified, it parses into `:wild`.

19.3.1.1.6 Wildcard Words in a Logical Pathname Namestring

Each *asterisk* in a *wildcard-word* matches a sequence of zero or more characters. The *wildcard-word* "`*`" parses into `:wild`; other *wildcard-words* parse into *strings*.

19.3.1.1.7 Lowercase Letters in a Logical Pathname Namestring

When parsing *words* and *wildcard-words*, lowercase letters are translated to uppercase.

19.3.1.1.8 Other Syntax in a Logical Pathname Namestring

The consequences of using characters other than those specified here in a *logical pathname namestring* are unspecified.

The consequences of using any value not specified here as a *logical pathname* component are unspecified.

19.3.2 Logical Pathname Components

19.3.2.1 Unspecific Components of a Logical Pathname

The device component of a *logical pathname* is always `:unspecific`; no other component of a *logical pathname* can be `:unspecific`.

19.3.2.2 Null Strings as Components of a Logical Pathname

The null string, `""`, is not a valid value for any component of a *logical pathname*.

pathname *System Class*

Class Precedence List:

pathname, **t**

Description:

A *pathname* is a structured *object* which represents a *filename*.

There are two kinds of *pathnames*—*physical pathnames* and *logical pathnames*.

logical-pathname *System Class*

Class Precedence List:

logical-pathname, **pathname**, **t**

Description:

A *pathname* that uses a *namestring* syntax that is *implementation-independent*, and that has component values that are *implementation-independent*. *Logical pathnames* do not refer directly to *filenames*

See Also:

Section 20.1 (File System Concepts), Section 2.4.8.14 (Sharpsign P), Section 22.1.3.11 (Printing Pathnames)

pathname *Function*

Syntax:

pathname *pathspec* → *pathname*

Arguments and Values:

pathspec—a *pathname designator*.

pathname—a *pathname*.

Description:

Returns the *pathname* denoted by *pathspec*.

pathname

If the *pathspec designator* is a *stream*, the *stream* can be either open or closed; in both cases, the **pathname** returned corresponds to the *filename* used to open the *file*. **pathname** returns the same *pathname* for a *file stream* after it is closed as it did when it was open.

If the *pathspec designator* is a *file stream* created by opening a *logical pathname*, a *logical pathname* is returned.

Examples:

```
;; There is a great degree of variability permitted here.  The next
;; several examples are intended to illustrate just a few of the many
;; possibilities.  Whether the name is canonicalized to a particular
;; case (either upper or lower) depends on both the file system and the
;; implementation since two different implementations using the same
;; file system might differ on many issues.  How information is stored
;; internally (and possibly presented in #S notation) might vary,
;; possibly requiring ‘accessors’ such as PATHNAME-NAME to perform case
;; conversion upon access.  The format of a namestring is dependent both
;; on the file system and the implementation since, for example, one
;; implementation might include the host name in a namestring, and
;; another might not.  #S notation would generally only be used in a
;; situation where no appropriate namestring could be constructed for use
;; with #P.
(setq p1 (pathname "test"))
→ #P"CHOCOLATE:TEST" ; with case canonicalization (e.g., VMS)
or→ #P"VANILLA:test"   ; without case canonicalization (e.g., Unix)
or→ #P"test"
or→ #S(PATHNAME :HOST "STRAWBERRY" :NAME "TEST")
or→ #S(PATHNAME :HOST "BELGIAN-CHOCOLATE" :NAME "test")
(setq p2 (pathname "test"))
→ #P"CHOCOLATE:TEST"
or→ #P"VANILLA:test"
or→ #P"test"
or→ #S(PATHNAME :HOST "STRAWBERRY" :NAME "TEST")
or→ #S(PATHNAME :HOST "BELGIAN-CHOCOLATE" :NAME "test")
(pathnamep p1) → true
(eq p1 (pathname p1)) → true
(eq p1 p2)
→ true
or→ false
(with-open-file (stream "test" :direction :output)
  (pathname stream))
→ #P"ORANGE-CHOCOLATE:>Gus>test.lisp.newest"
```

See Also:

pathname, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as

Filenames)

make-pathname *Function*

Syntax:

make-pathname `&key` *host device directory name type version defaults case*
→ *pathname*

Arguments and Values:

host—a *valid physical pathname host*. Complicated defaulting behavior; see below.

device—a *valid pathname device*. Complicated defaulting behavior; see below.

directory—a *valid pathname directory*. Complicated defaulting behavior; see below.

name—a *valid pathname name*. Complicated defaulting behavior; see below.

type—a *valid pathname type*. Complicated defaulting behavior; see below.

version—a *valid pathname version*. Complicated defaulting behavior; see below.

defaults—a *pathname designator*. The default is a *pathname* whose host component is the same as the host component of the *value* of ***default-pathname-defaults***, and whose other components are all **nil**.

case—one of `:common` or `:local`. The default is `:local`.

pathname—a *pathname*.

Description:

Constructs and returns a *pathname* from the supplied keyword arguments.

After the components supplied explicitly by *host*, *device*, *directory*, *name*, *type*, and *version* are filled in, the merging rules used by **merge-pathnames** are used to fill in any unsupplied components from the defaults supplied by *defaults*.

Whenever a *pathname* is constructed the components may be canonicalized if appropriate. For the explanation of the arguments that can be supplied for each component, see Section 19.2.1 (Pathname Components).

If *case* is supplied, it is treated as described in Section 19.2.2.1.2 (Case in Pathname Components).

The resulting *pathname* is a *logical pathname* if and only its host component is a *logical host* or a *string* that names a defined *logical host*.

make-pathname

If the *directory* is a *string*, it should be the name of a top level directory, and should not contain any punctuation characters; that is, specifying a *string*, *str*, is equivalent to specifying the list `(:absolute` *str*`)`. Specifying the symbol `:wild` is equivalent to specifying the list `(:absolute :wild-inferiors)`, or `(:absolute :wild)` in a file system that does not support `:wild-inferiors`.

Examples:

```
 ;; Implementation A - an implementation with access to a single
 ;;  Unix file system.  This implementation happens to never display
 ;;  the 'host' information in a namestring, since there is only one host.
 (make-pathname :directory '(:absolute "public" "games")
                :name "chess" :type "db")
→ #P"/public/games/chess.db"
```

```
 ;; Implementation B - an implementation with access to one or more
 ;;  VMS file systems.  This implementation displays 'host' information
 ;;  in the namestring only when the host is not the local host.
 ;;  It uses a double colon to separate a host name from the host's local
 ;;  file name.
 (make-pathname :directory '(:absolute "PUBLIC" "GAMES")
                :name "CHESS" :type "DB")
→ #P"SYS$DISK:[PUBLIC.GAMES]CHESS.DB"
 (make-pathname :host "BOBBY"
                :directory '(:absolute "PUBLIC" "GAMES")
                :name "CHESS" :type "DB")
→ #P"BOBBY::SYS$DISK:[PUBLIC.GAMES]CHESS.DB"
```

```
 ;; Implementation C - an implementation with simultaneous access to
 ;;  multiple file systems from the same Lisp image.  In this
 ;;  implementation, there is a convention that any text preceding the
 ;;  first colon in a pathname namestring is a host name.
 (dolist (case '(:common :local))
   (dolist (host '("MY-LISPM" "MY-VAX" "MY-UNIX"))
     (print (make-pathname :host host :case case
                           :directory '(:absolute "PUBLIC" "GAMES")
                           :name "CHESS" :type "DB"))))
▷ #P"MY-LISPM:>public>games>chess.db"
▷ #P"MY-VAX:SYS$DISK:[PUBLIC.GAMES]CHESS.DB"
▷ #P"MY-UNIX:/public/games/chess.db"
▷ #P"MY-LISPM:>public>games>chess.db"
▷ #P"MY-VAX:SYS$DISK:[PUBLIC.GAMES]CHESS.DB"
▷ #P"MY-UNIX:/PUBLIC/GAMES/CHESS.DB"
→ NIL
```

Affected By:

The *file system*.

See Also:

merge-pathnames, **pathname**, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

Notes:

Portable programs should not supply `:unspecific` for any component. See Section 19.2.2.2.3 (:UNSPECIFIC as a Component Value).

pathnamep *Function*

Syntax:

pathnamep *object* → *generalized-boolean*

Arguments and Values:

object—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *object* is of *type* **pathname**; otherwise, returns *false*.

Examples:

```
 (setq q "test")  → "test"
 (pathnamep q) → false
 (setq q (pathname "test"))
→ #S(PATHNAME :HOST NIL :DEVICE NIL :DIRECTORY NIL :NAME "test" :TYPE NIL
       :VERSION NIL)
 (pathnamep q) → true
 (setq q (logical-pathname "SYS:SITE;FOO.SYSTEM"))
→ #P"SYS:SITE;FOO.SYSTEM"
 (pathnamep q) → true
```

Notes:

```
 (pathnamep object) ≡ (typep object 'pathname)
```

pathname-host, pathname-device, pathname-directory, pathname-name, pathname-type, pathname-version *Function*

Syntax:

pathname-host *pathname* `&key` *case* → *host*

pathname-device *pathname* `&key` *case* → *device*

pathname-directory *pathname* `&key` *case* → *directory*

pathname-name *pathname* `&key` *case* → *name*

pathname-type *pathname* `&key` *case* → *type*

pathname-version *pathname* → *version*

Arguments and Values:

pathname—a *pathname designator*.

case—one of `:local` or `:common`. The default is `:local`.

host—a *valid pathname host*.

device—a *valid pathname device*.

directory—a *valid pathname directory*.

name—a *valid pathname name*.

type—a *valid pathname type*.

version—a *valid pathname version*.

Description:

These functions return the components of *pathname*.

If the *pathname designator* is a *pathname*, it represents the name used to open the file. This may be, but is not required to be, the actual name of the file.

If *case* is supplied, it is treated as described in Section 19.2.2.1.2 (Case in Pathname Components).

Examples:

```
 (setq q (make-pathname :host "KATHY"
                        :directory "CHAPMAN"
                        :name "LOGIN" :type "COM"))
→ #P"KATHY::[CHAPMAN]LOGIN.COM"
 (pathname-host q) → "KATHY"
 (pathname-name q) → "LOGIN"
 (pathname-type q) → "COM"

 ;; Because namestrings are used, the results shown in the remaining
 ;; examples are not necessarily the only possible results.  Mappings
 ;; from namestring representation to pathname representation are
 ;; dependent both on the file system involved and on the implementation
 ;; (since there may be several implementations which can manipulate the
 ;; the same file system, and those implementations are not constrained
 ;; to agree on all details). Consult the documentation for each
 ;; implementation for specific information on how namestrings are treated
 ;; that implementation.

 ;; VMS
 (pathname-directory (parse-namestring "[FOO.*.BAR]BAZ.LSP"))
→ (:ABSOLUTE "FOO" "BAR")
 (pathname-directory (parse-namestring "[FOO.*.BAR]BAZ.LSP") :case :common)
→ (:ABSOLUTE "FOO" "BAR")

 ;; Unix
 (pathname-directory "foo.l") → NIL
 (pathname-device "foo.l") → :UNSPECIFIC
 (pathname-name "foo.l") → "foo"
 (pathname-name "foo.l" :case :local) → "foo"
 (pathname-name "foo.l" :case :common) → "FOO"
 (pathname-type "foo.l") → "l"
 (pathname-type "foo.l" :case :local) → "l"
 (pathname-type "foo.l" :case :common) → "L"
 (pathname-type "foo") → :UNSPECIFIC
 (pathname-type "foo" :case :common) → :UNSPECIFIC
 (pathname-type "foo.") → ""
 (pathname-type "foo." :case :common) → ""
 (pathname-directory (parse-namestring "/foo/bar/baz.lisp") :case :local)
→ (:ABSOLUTE "foo" "bar")
 (pathname-directory (parse-namestring "/foo/bar/baz.lisp") :case :local)
→ (:ABSOLUTE "FOO" "BAR")
 (pathname-directory (parse-namestring "../baz.lisp"))
→ (:RELATIVE :UP)
 (PATHNAME-DIRECTORY (PARSE-NAMESTRING "/foo/BAR/../Mum/baz"))
→ (:ABSOLUTE "foo" "BAR" :UP "Mum")
```

```
 (PATHNAME-DIRECTORY (PARSE-NAMESTRING "/foo/BAR/../Mum/baz") :case :common)
→ (:ABSOLUTE "FOO" "bar" :UP "Mum")
 (PATHNAME-DIRECTORY (PARSE-NAMESTRING "/foo/*/bar/baz.l"))
→ (:ABSOLUTE "foo" :WILD "bar")
 (PATHNAME-DIRECTORY (PARSE-NAMESTRING "/foo/*/bar/baz.l") :case :common)
→ (:ABSOLUTE "FOO" :WILD "BAR")

 ;; Symbolics LMFS
 (pathname-directory (parse-namestring ">foo>**>bar>baz.lisp"))
→ (:ABSOLUTE "foo" :WILD-INFERIORS "bar")
 (pathname-directory (parse-namestring ">foo>*>bar>baz.lisp"))
→ (:ABSOLUTE "foo" :WILD "bar")
 (pathname-directory (parse-namestring ">foo>*>bar>baz.lisp") :case :common)
→ (:ABSOLUTE "FOO" :WILD "BAR")
 (pathname-device (parse-namestring ">foo>baz.lisp")) → :UNSPECIFIC
```

Affected By:

The *implementation* and the host *file system*.

Exceptional Situations:

Should signal an error of *type* **type-error** if its first argument is not a *pathname*.

See Also:

pathname, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

load-logical-pathname-translations *Function*

Syntax:

load-logical-pathname-translations *host* → *just-loaded*

Arguments and Values:

host—a *string*.

just-loaded—a *generalized boolean*.

Description:

Searches for and loads the definition of a *logical host* named *host*, if it is not already defined. The specific nature of the search is *implementation-defined*.

If the *host* is already defined, no attempt to find or load a definition is attempted, and *false* is returned. If the *host* is not already defined, but a definition is successfully found and loaded, *true* is returned. Otherwise, an error is signaled.

Examples:

```
 (translate-logical-pathname "hacks:weather;barometer.lisp.newest")
▷ Error: The logical host HACKS is not defined.
 (load-logical-pathname-translations "HACKS")
▷ ;; Loading SYS:SITE;HACKS.TRANSLATIONS
▷ ;; Loading done.
→ true
 (translate-logical-pathname "hacks:weather;barometer.lisp.newest")
→ #P"HELIUM:[SHARED.HACKS.WEATHER]BAROMETER.LSP;0"
 (load-logical-pathname-translations "HACKS")
→ false
```

Exceptional Situations:

If no definition is found, an error of *type* **error** is signaled.

See Also:

logical-pathname

Notes:

Logical pathname definitions will be created not just by *implementors* but also by *programmers*. As such, it is important that the search strategy be documented. For example, an *implementation* might define that the definition of a *host* is to be found in a file called "*host*.translations" in some specifically named directory.

logical-pathname-translations *Accessor*

Syntax:

logical-pathname-translations *host* → *translations*

(setf (logical-pathname-translations *host*) *new-translations*)

Arguments and Values:

host–a *logical host designator*.

translations, *new-translations*—a *list*.

logical-pathname-translations

Description:

Returns the host's *list* of translations. Each translation is a *list* of at least two elements: *from-wildcard* and *to-wildcard*. Any additional elements are *implementation-defined*. *From-wildcard* is a *logical pathname* whose host is *host*. *To-wildcard* is a *pathname*.

`(setf (logical-pathname-translations` *host*`)` *translations*`)` sets a *logical pathname* host's *list* of *translations*. If *host* is a *string* that has not been previously used as a *logical pathname* host, a new *logical pathname* host is defined; otherwise an existing host's translations are replaced. *logical pathname* host names are compared with **string-equal**.

When setting the translations list, each *from-wildcard* can be a *logical pathname* whose host is *host* or a *logical pathname* namestring parseable by `(parse-namestring` *string host*`)`, where *host* represents the appropriate *object* as defined by **parse-namestring**. Each *to-wildcard* can be anything coercible to a *pathname* by `(pathname` *to-wildcard*`)`. If *to-wildcard* coerces to a *logical pathname*, **translate-logical-pathname** will perform repeated translation steps when it uses it.

host is either the host component of a *logical pathname* or a *string* that has been defined as a *logical pathname* host name by **setf** of **logical-pathname-translations**.

Examples:

```
;;;A very simple example of setting up a logical pathname host.  No
;;;translations are necessary to get around file system restrictions, so
;;;all that is necessary is to specify the root of the physical directory
;;;tree that contains the logical file system.
;;;The namestring syntax on the right-hand side is implementation-dependent.
(setf (logical-pathname-translations "foo")
      '(("**;*.*.*"              "MY-LISPM:>library>foo>**>")))


;;;Sample use of that logical pathname.  The return value
;;;is implementation-dependent.
(translate-logical-pathname "foo:bar;baz;mum.quux.3")
→ #P"MY-LISPM:>library>foo>bar>baz>mum.quux.3"


;;;A more complex example, dividing the files among two file servers
;;;and several different directories.  This Unix doesn't support
;;;:WILD-INFERIORS in the directory, so each directory level must
;;;be translated individually.  No file name or type translations
;;;are required except for .MAIL to .MBX.
;;;The namestring syntax on the right-hand side is implementation-dependent.
(setf (logical-pathname-translations "prog")
      '(("RELEASED;*.*.*"        "MY-UNIX:/sys/bin/my-prog/")
        ("RELEASED;*;*.*.*"      "MY-UNIX:/sys/bin/my-prog/*/")
        ("EXPERIMENTAL;*.*.*"    "MY-UNIX:/usr/Joe/development/prog/")
        ("EXPERIMENTAL;DOCUMENTATION;*.*.*"
                                 "MY-VAX:SYS$DISK:[JOE.DOC]")
        ("EXPERIMENTAL;*;*.*.*"  "MY-UNIX:/usr/Joe/development/prog/*/")
        ("MAIL;**;*.MAIL"        "MY-VAX:SYS$DISK:[JOE.MAIL.PROG...]*.MBX")))


;;;Sample use of that logical pathname.  The return value
;;;is implementation-dependent.
(translate-logical-pathname "prog:mail;save;ideas.mail.3")
→ #P"MY-VAX:SYS$DISK:[JOE.MAIL.PROG.SAVE]IDEAS.MBX.3"


;;;Example translations for a program that uses three files main.lisp,
;;;auxiliary.lisp, and documentation.lisp.  These translations might be
;;;supplied by a software supplier as examples.


;;;For Unix with long file names
(setf (logical-pathname-translations "prog")
      '(("CODE;*.*.*"             "/lib/prog/")))


;;;Sample use of that logical pathname.  The return value
;;;is implementation-dependent.
(translate-logical-pathname "prog:code;documentation.lisp")
→ #P"/lib/prog/documentation.lisp"


;;;For Unix with 14-character file names, using .lisp as the type
(setf (logical-pathname-translations "prog")
      '(("CODE;DOCUMENTATION.*.*" "/lib/prog/docum.*")
        ("CODE;*.*.*"             "/lib/prog/")))

;;;Sample use of that logical pathname.  The return value
;;;is implementation-dependent.
(translate-logical-pathname "prog:code;documentation.lisp")
→ #P"/lib/prog/docum.lisp"


;;;For Unix with 14-character file names, using .l as the type
;;;The second translation shortens the compiled file type to .b
(setf (logical-pathname-translations "prog")
      `(("**;*.LISP.*"            ,(logical-pathname "PROG:**;*.L.*"))
        (,(compile-file-pathname (logical-pathname "PROG:**;*.LISP.*"))
```

logical-pathname-translations

```
                               ,(logical-pathname "PROG:**;*.B.*"))
         ("CODE;DOCUMENTATION.*.*" "/lib/prog/documentatio.*")
         ("CODE;*.*.*"             "/lib/prog/")))


 ;;;Sample use of that logical pathname.  The return value
 ;;;is implementation-dependent.
 (translate-logical-pathname "prog:code;documentation.lisp")
→ #P"/lib/prog/documentatio.l"


 ;;;For a Cray with 6 character names and no directories, types, or versions.
 (setf (logical-pathname-translations "prog")
       (let ((l '(("MAIN" "PGMN")
                  ("AUXILIARY" "PGAUX")
                  ("DOCUMENTATION" "PGDOC")))
             (logpath (logical-pathname "prog:code;"))
             (phypath (pathname "XXX")))
         (append
           ;; Translations for source files
           (mapcar #'(lambda (x)
                       (let ((log (first x))
                             (phy (second x)))
                         (list (make-pathname :name log
                                              :type "LISP"
                                              :version :wild
                                              :defaults logpath)
                               (make-pathname :name phy
                                              :defaults phypath))))
                   l)
           ;; Translations for compiled files
           (mapcar #'(lambda (x)
                       (let* ((log (first x))
                              (phy (second x))
                              (com (compile-file-pathname
                                     (make-pathname :name log
                                                    :type "LISP"
                                                    :version :wild
                                                    :defaults logpath))))
                         (setq phy (concatenate 'string phy "B"))
                         (list com
                               (make-pathname :name phy
                                              :defaults phypath))))
                   l))))
 ;;;Sample use of that logical pathname.  The return value
 ;;;is implementation-dependent.
 (translate-logical-pathname "prog:code;documentation.lisp")
→ #P"PGDOC"
```

Exceptional Situations:

If *host* is incorrectly supplied, an error of *type* **type-error** is signaled.

See Also:

logical-pathname, Section 19.1.2 (Pathnames as Filenames)

Notes:

Implementations can define additional *functions* that operate on *logical pathname* hosts, for example to specify additional translation rules or options.

logical-pathname *Function*

Syntax:

logical-pathname *pathspec* → *logical-pathname*

Arguments and Values:

pathspec—a *logical pathname*, a *logical pathname namestring*, or a *stream*.

logical-pathname—a *logical pathname*.

Description:

logical-pathname converts *pathspec* to a *logical pathname* and returns the new *logical pathname*. If *pathspec* is a *logical pathname namestring*, it should contain a host component and its following *colon*. If *pathspec* is a *stream*, it should be one for which **pathname** returns a *logical pathname*.

If *pathspec* is a *stream*, the *stream* can be either open or closed. **logical-pathname** returns the same *logical pathname* after a file is closed as it did when the file was open. It is an error if *pathspec* is a *stream* that is created with **make-two-way-stream**, **make-echo-stream**, **make-broadcast-stream**, **make-concatenated-stream**, **make-string-input-stream**, or **make-string-output-stream**.

Exceptional Situations:

Signals an error of *type* **type-error** if *pathspec* isn't supplied correctly.

See Also:

logical-pathname, **translate-logical-pathname**, Section 19.3 (Logical Pathnames)

∗default-pathname-defaults∗ *Variable*

Value Type:

a *pathname object*.

Initial Value:

An *implementation-dependent pathname*, typically in the working directory that was current when Common Lisp was started up.

Description:

a *pathname*, used as the default whenever a *function* needs a default *pathname* and one is not supplied.

Examples:

```
 ;; This example illustrates a possible usage for a hypothetical Lisp running on a
 ;; DEC TOPS-20 file system.  Since pathname conventions vary between Lisp
 ;; implementations and host file system types, it is not possible to provide a
 ;; general-purpose, conforming example.
 *default-pathname-defaults* → #P"PS:<FRED>"
 (merge-pathnames (make-pathname :name "CALENDAR"))
→ #P"PS:<FRED>CALENDAR"
 (let ((*default-pathname-defaults* (pathname "<MARY>")))
   (merge-pathnames (make-pathname :name "CALENDAR")))
→ #P"<MARY>CALENDAR"
```

Affected By:

The *implementation*.

namestring, file-namestring, directory-namestring, host-namestring, enough-namestring *Function*

Syntax:

namestring *pathname* → *namestring*

file-namestring *pathname* → *namestring*

directory-namestring *pathname* → *namestring*

host-namestring *pathname* → *namestring*

enough-namestring *pathname* `&optional` *defaults* → *namestring*

Arguments and Values:

pathname—a *pathname designator*.

defaults—a *pathname designator*. The default is the *value* of ***default-pathname-defaults***.

namestring—a *string* or **nil**.

Description:

These functions convert *pathname* into a namestring. The name represented by *pathname* is returned as a *namestring* in an *implementation-dependent* canonical form.

namestring returns the full form of *pathname*.

file-namestring returns just the name, type, and version components of *pathname*.

directory-namestring returns the directory name portion.

host-namestring returns the host name.

enough-namestring returns an abbreviated namestring that is just sufficient to identify the file named by *pathname* when considered relative to the *defaults*. It is required that

```
 (merge-pathnames (enough-namestring pathname defaults) defaults)
≡ (merge-pathnames (parse-namestring pathname nil defaults) defaults)
```

in all cases, and the result of **enough-namestring** is the shortest reasonable *string* that will satisfy this criterion.

It is not necessarily possible to construct a valid *namestring* by concatenating some of the three shorter *namestrings* in some order.

Examples:

```
 (namestring "getty")
→ "getty"
 (setq q (make-pathname :host "kathy"
                        :directory
                          (pathname-directory *default-pathname-defaults*)
                        :name "getty"))
→ #S(PATHNAME :HOST "kathy" :DEVICE NIL :DIRECTORY directory-name
       :NAME "getty" :TYPE NIL :VERSION NIL)
 (file-namestring q) → "getty"
 (directory-namestring q) → directory-name
 (host-namestring q) → "kathy"


 ;;;Using Unix syntax and the wildcard conventions used by the
 ;;;particular version of Unix on which this example was created:
```

```
 (namestring
   (translate-pathname "/usr/dmr/hacks/frob.l"
                       "/usr/d*/hacks/*.l"
                       "/usr/d*/backup/hacks/backup-*.*"))
→ "/usr/dmr/backup/hacks/backup-frob.l"
 (namestring
   (translate-pathname "/usr/dmr/hacks/frob.l"
                       "/usr/d*/hacks/fr*.l"
                       "/usr/d*/backup/hacks/backup-*.*"))
→ "/usr/dmr/backup/hacks/backup-ob.l"

 ;;;This is similar to the above example but uses two different hosts,
 ;;;U: which is a Unix and V: which is a VMS.  Note the translation
 ;;;of file type and alphabetic case conventions.
 (namestring
   (translate-pathname "U:/usr/dmr/hacks/frob.l"
                       "U:/usr/d*/hacks/*.l"
                       "V:SYS$DISK:[D*.BACKUP.HACKS]BACKUP-*.*"))
→ "V:SYS$DISK:[DMR.BACKUP.HACKS]BACKUP-FROB.LSP"
 (namestring
   (translate-pathname "U:/usr/dmr/hacks/frob.l"
                       "U:/usr/d*/hacks/fr*.l"
                       "V:SYS$DISK:[D*.BACKUP.HACKS]BACKUP-*.*"))
→ "V:SYS$DISK:[DMR.BACKUP.HACKS]BACKUP-OB.LSP"
```

See Also:

truename, **merge-pathnames**, **pathname**, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

parse-namestring *Function*

Syntax:

parse-namestring *thing* `&optional` *host default-pathname* `&key` *start end junk-allowed*
→ *pathname, position*

Arguments and Values:

thing—a *string*, a *pathname*, or a *stream associated with a file*.

host—a *valid pathname host*, a *logical host*, or **nil**.

default-pathname—a *pathname designator*. The default is the *value* of ***default-pathname-defaults***.

parse-namestring

start, *end*—*bounding index designators* of *thing*. The defaults for *start* and *end* are 0 and **nil**, respectively.

junk-allowed—a *generalized boolean*. The default is *false*.

pathname—a *pathname*, or **nil**.

position—a *bounding index designator* for *thing*.

Description:

Converts *thing* into a *pathname*.

The *host* supplies a host name with respect to which the parsing occurs.

If *thing* is a *stream associated with a file*, processing proceeds as if the *pathname* used to open that *file* had been supplied instead.

If *thing* is a *pathname*, the *host* and the host component of *thing* are compared. If they match, two values are immediately returned: *thing* and *start*; otherwise (if they do not match), an error is signaled.

Otherwise (if *thing* is a *string*), **parse-namestring** parses the name of a *file* within the substring of *thing* bounded by *start* and *end*.

If *thing* is a *string* then the substring of *thing bounded* by *start* and *end* is parsed into a *pathname* as follows:

- If *host* is a *logical host* then *thing* is parsed as a *logical pathname namestring* on the *host*.
- If *host* is **nil** and *thing* is a syntactically valid *logical pathname namestring* containing an explicit host, then it is parsed as a *logical pathname namestring*.
- If *host* is **nil**, *default-pathname* is a *logical pathname*, and *thing* is a syntactically valid *logical pathname namestring* without an explicit host, then it is parsed as a *logical pathname namestring* on the host that is the host component of *default-pathname*.
- Otherwise, the parsing of *thing* is *implementation-defined*.

In the first of these cases, the host portion of the *logical pathname* namestring and its following *colon* are optional.

If the host portion of the namestring and *host* are both present and do not match, an error is signaled.

If *junk-allowed* is *true*, then the *primary value* is the *pathname* parsed or, if no syntactically correct *pathname* was seen, **nil**. If *junk-allowed* is *false*, then the entire substring is scanned, and the *primary value* is the *pathname* parsed.

In either case, the *secondary value* is the index into *thing* of the delimiter that terminated the parse, or the index beyond the substring if the parse terminated at the end of the substring (as will always be the case if *junk-allowed* is *false*).

Parsing a *null string* always succeeds, producing a *pathname* with all components (except the host) equal to **nil**.

If *thing* contains an explicit host name and no explicit device name, then it is *implementation-defined* whether **parse-namestring** will supply the standard default device for that host as the device component of the resulting *pathname*.

Examples:

```
 (setq q (parse-namestring "test"))
→ #S(PATHNAME :HOST NIL :DEVICE NIL :DIRECTORY NIL :NAME "test"
       :TYPE NIL :VERSION NIL)
 (pathnamep q) → true
 (parse-namestring "test")
→ #S(PATHNAME :HOST NIL :DEVICE NIL :DIRECTORY NIL :NAME "test"
       :TYPE NIL :VERSION NIL), 4
 (setq s (open xxx)) → #<Input File Stream...>
 (parse-namestring s)
→ #S(PATHNAME :HOST NIL :DEVICE NIL :DIRECTORY NIL :NAME xxx
       :TYPE NIL :VERSION NIL), 0
 (parse-namestring "test" nil nil :start 2 :end 4 )
 → #S(PATHNAME ...), 15
 (parse-namestring "foo.lisp")
→ #P"foo.lisp"
```

Exceptional Situations:

If *junk-allowed* is *false*, an error of *type* **parse-error** is signaled if *thing* does not consist entirely of the representation of a *pathname*, possibly surrounded on either side by *whitespace*$_1$ characters if that is appropriate to the cultural conventions of the implementation.

If *host* is supplied and not **nil**, and *thing* contains a manifest host name, an error of *type* **error** is signaled if the hosts do not match.

If *thing* is a *logical pathname* namestring and if the host portion of the namestring and *host* are both present and do not match, an error of *type* **error** is signaled.

See Also:

pathname, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.2.2.2.3 (:UNSPECIFIC as a Component Value), Section 19.1.2 (Pathnames as Filenames)

wild-pathname-p *Function*

Syntax:

wild-pathname-p *pathname* `&optional` *field-key* → *generalized-boolean*

Arguments and Values:

pathname—a *pathname designator*.

Field-key—one of `:host`, `:device` `:directory`, `:name`, `:type`, `:version`, or **nil**.

generalized-boolean—a *generalized boolean*.

Description:

wild-pathname-p tests *pathname* for the presence of wildcard components.

If *pathname* is a *pathname* (as returned by **pathname**) it represents the name used to open the file. This may be, but is not required to be, the actual name of the file.

If *field-key* is not supplied or **nil**, **wild-pathname-p** returns true if *pathname* has any wildcard components, **nil** if *pathname* has none. If *field-key* is *non-nil*, **wild-pathname-p** returns true if the indicated component of *pathname* is a wildcard, **nil** if the component is not a wildcard.

Examples:

```
;;;The following examples are not portable.  They are written to run
;;;with particular file systems and particular wildcard conventions.
;;;Other implementations will behave differently.  These examples are
;;;intended to be illustrative, not to be prescriptive.

(wild-pathname-p (make-pathname :name :wild)) → true
(wild-pathname-p (make-pathname :name :wild) :name) → true
(wild-pathname-p (make-pathname :name :wild) :type) → false
(wild-pathname-p (pathname "s:>foo>**>")) → true ;Lispm
(wild-pathname-p (pathname :name "F*O")) → true ;Most places
```

Exceptional Situations:

If *pathname* is not a *pathname*, a *string*, or a *stream associated with a file* an error of *type* **type-error** is signaled.

See Also:

pathname, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

Notes:

Not all implementations support wildcards in all fields. See Section 19.2.2.2.2 (:WILD as a

Component Value) and Section 19.2.2.3 (Restrictions on Wildcard Pathnames).

pathname-match-p *Function*

Syntax:

pathname-match-p *pathname wildcard* → *generalized-boolean*

Arguments and Values:

pathname—a *pathname designator*.

wildcard—a *designator* for a *wild pathname*.

generalized-boolean—a *generalized boolean*.

Description:

pathname-match-p returns true if *pathname* matches *wildcard*, otherwise **nil**. The matching rules are *implementation-defined* but should be consistent with **directory**. Missing components of *wildcard* default to `:wild`.

It is valid for *pathname* to be a wild *pathname*; a wildcard field in *pathname* only matches a wildcard field in *wildcard* (*i.e.*, **pathname-match-p** is not commutative). It is valid for *wildcard* to be a non-wild *pathname*.

Exceptional Situations:

If *pathname* or *wildcard* is not a *pathname*, *string*, or *stream associated with a file* an error of *type* **type-error** is signaled.

See Also:

directory, **pathname**, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

translate-logical-pathname *Function*

Syntax:

translate-logical-pathname *pathname* `&key` → *physical-pathname*

Arguments and Values:

pathname—a *pathname designator*, or a *logical pathname namestring*.

physical-pathname—a *physical pathname*.

Description:

Translates *pathname* to a *physical pathname*, which it returns.

If *pathname* is a *stream*, the *stream* can be either open or closed. **translate-logical-pathname** returns the same physical pathname after a file is closed as it did when the file was open. It is an error if *pathname* is a *stream* that is created with **make-two-way-stream**, **make-echo-stream**, **make-broadcast-stream**, **make-concatenated-stream**, **make-string-input-stream**, **make-string-output-stream**.

If *pathname* is a *logical pathname* namestring, the host portion of the *logical pathname* namestring and its following *colon* are required.

Pathname is first coerced to a *pathname*. If the coerced *pathname* is a physical pathname, it is returned. If the coerced *pathname* is a *logical pathname*, the first matching translation (according to **pathname-match-p**) of the *logical pathname* host is applied, as if by calling **translate-pathname**. If the result is a *logical pathname*, this process is repeated. When the result is finally a physical pathname, it is returned. If no translation matches, an error is signaled.

translate-logical-pathname might perform additional translations, typically to provide translation of file types to local naming conventions, to accomodate physical file systems with limited length names, or to deal with special character requirements such as translating hyphens to underscores or uppercase letters to lowercase. Any such additional translations are *implementation-defined*. Some implementations do no additional translations.

There are no specified keyword arguments for **translate-logical-pathname**, but implementations are permitted to extend it by adding keyword arguments.

Examples:

See **logical-pathname-translations**.

Exceptional Situations:

If *pathname* is incorrectly supplied, an error of *type* **type-error** is signaled.

If no translation matches, an error of *type* **file-error** is signaled.

See Also:

logical-pathname, **logical-pathname-translations**, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

translate-pathname *Function*

Syntax:

translate-pathname *source from-wildcard to-wildcard* `&key`
  → *translated-pathname*

Arguments and Values:

source—a *pathname designator*.

from-wildcard—a *pathname designator*.

to-wildcard—a *pathname designator*.

translated-pathname—a *pathname*.

Description:

translate-pathname translates *source* (that matches *from-wildcard*) into a corresponding *pathname* that matches *to-wildcard*, and returns the corresponding *pathname*.

The resulting *pathname* is *to-wildcard* with each wildcard or missing field replaced by a portion of *source*. A "wildcard field" is a *pathname* component with a value of `:wild`, a `:wild` element of a *list*-valued directory component, or an *implementation-defined* portion of a component, such as the `"*"` in the complex wildcard string `"foo*bar"` that some implementations support. An implementation that adds other wildcard features, such as regular expressions, must define how **translate-pathname** extends to those features. A "missing field" is a *pathname* component with a value of **nil**.

The portion of *source* that is copied into the resulting *pathname* is *implementation-defined*. Typically it is determined by the user interface conventions of the file systems involved. Usually it is the portion of *source* that matches a wildcard field of *from-wildcard* that is in the same position as the wildcard or missing field of *to-wildcard*. If there is no wildcard field in *from-wildcard* at that position, then usually it is the entire corresponding *pathname* component of *source*, or in the case of a *list*-valued directory component, the entire corresponding *list* element.

During the copying of a portion of *source* into the resulting *pathname*, additional *implementation-defined* translations of *case* or file naming conventions might occur, especially when *from-wildcard* and *to-wildcard* are for different hosts.

It is valid for *source* to be a wild *pathname*; in general this will produce a wild result. It is valid for *from-wildcard* and/or *to-wildcard* to be non-wild *pathnames*.

There are no specified keyword arguments for **translate-pathname**, but implementations are permitted to extend it by adding keyword arguments.

translate-pathname maps customary case in *source* into customary case in the output *pathname*.

Examples:

```
 ;; The results of the following five forms are all implementation-dependent.
 ;; The second item in particular is shown with multiple results just to
 ;; emphasize one of many particular variations which commonly occurs.
 (pathname-name (translate-pathname "foobar" "foo*" "*baz")) → "barbaz"
 (pathname-name (translate-pathname "foobar" "foo*" "*"))
→ "foobar"
or
→ "bar"
 (pathname-name (translate-pathname "foobar" "*"    "foo*")) → "foofoobar"
 (pathname-name (translate-pathname "bar"    "*"    "foo*")) → "foobar"
 (pathname-name (translate-pathname "foobar" "foo*" "baz*")) → "bazbar"

 (defun translate-logical-pathname-1 (pathname rules)
   (let ((rule (assoc pathname rules :test #'pathname-match-p)))
     (unless rule (error "No translation rule for ~A" pathname))
     (translate-pathname pathname (first rule) (second rule))))
 (translate-logical-pathname-1 "FOO:CODE;BASIC.LISP"
                      '(("FOO:DOCUMENTATION;" "MY-UNIX:/doc/foo/")
                        ("FOO:CODE;"          "MY-UNIX:/lib/foo/")
                        ("FOO:PATCHES;*;"     "MY-UNIX:/lib/foo/patch/*/")))
→ #P"MY-UNIX:/lib/foo/basic.l"

;;;This example assumes one particular set of wildcard conventions
;;;Not all file systems will run this example exactly as written
 (defun rename-files (from to)
   (dolist (file (directory from))
     (rename-file file (translate-pathname file from to))))
 (rename-files "/usr/me/*.lisp" "/dev/her/*.l")
   ;Renames /usr/me/init.lisp to /dev/her/init.l
 (rename-files "/usr/me/pcl*/*" "/sys/pcl/*/")
   ;Renames /usr/me/pcl-5-may/low.lisp to /sys/pcl/pcl-5-may/low.lisp
   ;In some file systems the result might be /sys/pcl/5-may/low.lisp
 (rename-files "/usr/me/pcl*/*" "/sys/library/*/")
   ;Renames /usr/me/pcl-5-may/low.lisp to /sys/library/pcl-5-may/low.lisp
   ;In some file systems the result might be /sys/library/5-may/low.lisp
 (rename-files "/usr/me/foo.bar" "/usr/me2/")
   ;Renames /usr/me/foo.bar to /usr/me2/foo.bar
 (rename-files "/usr/joe/*-recipes.text" "/usr/jim/cookbook/joe's-*-rec.text")
   ;Renames /usr/joe/lamb-recipes.text to /usr/jim/cookbook/joe's-lamb-rec.text
   ;Renames /usr/joe/pork-recipes.text to /usr/jim/cookbook/joe's-pork-rec.text
```

```
;Renames /usr/joe/veg-recipes.text to /usr/jim/cookbook/joe's-veg-rec.text
```

Exceptional Situations:

If any of *source*, *from-wildcard*, or *to-wildcard* is not a *pathname*, a *string*, or a *stream associated with a file* an error of *type* **type-error** is signaled.

`(pathname-match-p` *source from-wildcard*`)` must be true or an error of *type* **error** is signaled.

See Also:

namestring, **pathname-host**, **pathname**, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

Notes:

The exact behavior of **translate-pathname** cannot be dictated by the Common Lisp language and must be allowed to vary, depending on the user interface conventions of the file systems involved.

The following is an implementation guideline. One file system performs this operation by examining each piece of the three *pathnames* in turn, where a piece is a *pathname* component or a *list* element of a structured component such as a hierarchical directory. Hierarchical directory elements in *from-wildcard* and *to-wildcard* are matched by whether they are wildcards, not by depth in the directory hierarchy. If the piece in *to-wildcard* is present and not wild, it is copied into the result. If the piece in *to-wildcard* is `:wild` or **nil**, the piece in *source* is copied into the result. Otherwise, the piece in *to-wildcard* might be a complex wildcard such as `"foo*bar"` and the piece in *from-wildcard* should be wild; the portion of the piece in *source* that matches the wildcard portion of the piece in *from-wildcard* replaces the wildcard portion of the piece in *to-wildcard* and the value produced is used in the result.

merge-pathnames *Function*

Syntax:

merge-pathnames *pathname* `&optional` *default-pathname default-version*
→ *merged-pathname*

Arguments and Values:

pathname—a *pathname designator*.

default-pathname—a *pathname designator*. The default is the *value* of ***default-pathname-defaults***.

default-version—a *valid pathname version*. The default is `:newest`.

merged-pathname—a *pathname*.

merge-pathnames

Description:

Constructs a *pathname* from *pathname* by filling in any unsupplied components with the corresponding values from *default-pathname* and *default-version*.

Defaulting of pathname components is done by filling in components taken from another *pathname*. This is especially useful for cases such as a program that has an input file and an output file. Unspecified components of the output pathname will come from the input pathname, except that the type should not default to the type of the input pathname but rather to the appropriate default type for output from the program; for example, see the *function* **compile-file-pathname**.

If no version is supplied, *default-version* is used. If *default-version* is **nil**, the version component will remain unchanged.

If *pathname* explicitly specifies a host and not a device, and if the host component of *default-pathname* matches the host component of *pathname*, then the device is taken from the *default-pathname*; otherwise the device will be the default file device for that host. If *pathname* does not specify a host, device, directory, name, or type, each such component is copied from *default-pathname*. If *pathname* does not specify a name, then the version, if not provided, will come from *default-pathname*, just like the other components. If *pathname* does specify a name, then the version is not affected by *default-pathname*. If this process leaves the version missing, the *default-version* is used. If the host's file name syntax provides a way to input a version without a name or type, the user can let the name and type default but supply a version different from the one in *default-pathname*.

If *pathname* is a *stream*, *pathname* effectively becomes `(pathname` *pathname*`)`. **merge-pathnames** can be used on either an open or a closed *stream*.

If *pathname* is a *pathname* it represents the name used to open the file. This may be, but is not required to be, the actual name of the file.

merge-pathnames recognizes a *logical pathname namestring* when *default-pathname* is a *logical pathname*, or when the *namestring* begins with the name of a defined *logical host* followed by a *colon*. In the first of these two cases, the host portion of the *logical pathname namestring* and its following *colon* are optional.

merge-pathnames returns a *logical pathname* if and only if its first argument is a *logical pathname*, or its first argument is a *logical pathname namestring* with an explicit host, or its first argument does not specify a host and the *default-pathname* is a *logical pathname*.

Pathname merging treats a relative directory specially. If `(pathname-directory` *pathname*`)` is a *list* whose *car* is `:relative`, and `(pathname-directory` *default-pathname*`)` is a *list*, then the merged directory is the value of

```
(append (pathname-directory default-pathname)
        (cdr  ;remove :relative from the front
          (pathname-directory pathname)))
```

except that if the resulting *list* contains a *string* or `:wild` immediately followed by `:back`,

merge-pathnames

both of them are removed. This removal of redundant `:back` *keywords* is repeated as many times as possible. If `(pathname-directory` *default-pathname*`)` is not a *list* or `(pathname-directory` *pathname*`)` is not a *list* whose *car* is `:relative`, the merged directory is `(or (pathname-directory` *pathname*`) (pathname-directory` *default-pathname*`))`

merge-pathnames maps customary case in *pathname* into customary case in the output *pathname*.

Examples:

```
 (merge-pathnames "CMUC::FORMAT"
                  "CMUC::PS:<LISPIO>.FASL")
→ #P"CMUC::PS:<LISPIO>FORMAT.FASL.0"
```

See Also:

default-pathname-defaults, **pathname**, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

Notes:

The net effect is that if just a name is supplied, the host, device, directory, and type will come from *default-pathname*, but the version will come from *default-version*. If nothing or just a directory is supplied, the name, type, and version will come from *default-pathname* together.