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*.

The next figure 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`

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>"
NOT=>  #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. The next figure 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. The next figure 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*.