24. Naming of Files

A Lisp Machine generally has access to many file systems. While it may have its own file system on its own disks, usually a community of Lisp Machine users want to have a shared file system accessible by any of the Lisp Machines over a network. These shared file systems can be implemented by any computer that is capable of providing file system service. A file server computer may be a special-purpose computer that does nothing but service file system requests from computers on a network, or it may be a time-sharing system.

Programs need to use names to designate files within these file systems. The main difficulty in dealing with names of files is that different file systems have different naming formats for files. For example, in the ITS file system, a typical name looks like:

```
DSK: GEORGE; FOO QFASL
```

with DSK being a device name, GEORGE being a directory name, FOO being the first file name and QFASL being the second file name. However, in TOPS-20, a similar file name is expressed as:

```
PS:<GEORGE>FOO.QFASL
```

It would be unreasonable for each program that deals with file names to be expected to know about each different file name format that exists, or new formats that could get added in the future. However, existing programs should retain their abilities to manipulate the names.

The functions and flavors described in this chapter exist to solve this problem. They provide an interface through which a program can deal with names of files and manipulate them without depending on anything about their syntax. This lets a program deal with multiple remote file servers simultaneously, using a uniform set of conventions.

24.1 Pathnames

All file systems dealt with by the Lisp Machine are mapped into a common model, in which files are named by something called a *pathname*. A pathname always has six components, each with a standard meaning. These components are the common interface that allows programs to work the same way with different file systems; the mapping of the pathname components into the concepts peculiar to each file system is taken care of by the pathname software. Pathname components are described in the following section, and the mappings between components and user syntax is described for each file system later in this chapter.

pathnamep *object*
t if *object* is a pathname.

A pathname is an instance of a flavor (see chapter 21, page 401); exactly which flavor depends on what the host of the pathname is, but **pathname** is always one of its component flavors. If *p* is a pathname, then (typep *p* 'pathname) returns t. One of the messages handled by host objects is the :pathname-flavor operation, which returns the name of the flavor to use for pathnames on that host. And one of the differences between host flavors is how they handle this operation.

There are functions for manipulating pathnames, and there are also messages that can be sent to them. These are described later in this chapter.

Two important operations of the pathname system are *parsing* and *merging*. Parsing is the conversion of a string—which might be something typed in by the user when asked to supply the name of a file—into a pathname object. This involves finding out what host the pathname is for, then using the file name syntax conventions of that host to parse the string into the standard pathname components. Merging is the operation that takes a pathname with missing components and supplies values for those components from a set of defaults.

The function **string**, applied to a pathname, converts it into a string that is in the file name syntax of its host's file system, except that the name of the host followed by a colon is inserted at the front. This is the inverse of parsing. **princ** of a pathname also does this, then prints the contents of the string. Flavor operations such as **:string-for-dired** exist which convert all or part of a pathname to a string in other fashions that are designed for specific applications. **prin1** of a pathname prints the pathname using the **#c** syntax so it can be read back in to produce an equivalent pathname (or the same pathname, if read in the same session).

Since each kind of file server can have its own character string representation of names of its files, there has to be a different parser for each of these representations, capable of examining such a character string and figuring out what each component is. The parsers all work differently. How can the parsing operation know which parser to use? The first thing that the parser does is to figure out which host this filename belongs to. A filename character string may specify a host explicitly by having the name of the host, followed by a colon, at either the beginning or the end of the string. For example, the following strings all specify hosts explicitly:

```
AI: COMMON; GEE WHIZ            ; This specifies host AI.
COMMON; GEE WHIZ AI:            ; So does this.
AI: ARC: USERS1; FOO BAR        ; So does this.
ARC: USERS1; FOO BAR AI:        ; So does this.
EE:PS:<COMMON>GEE.WHIZ.5        ; This specifies host EE.
PS:<COMMON>GEE.WHIZ.5 EE:       ; So does this.
```

If the string does not specify a host explicitly, the parser chooses a host by default and uses the syntax for that host. The optional arguments passed to the parsing function (**fs:parse-pathname**) tell it which host to assume. Note: the parser is not confused by strings starting with **DSK:** or **PS:** because it knows that neither of those is a valid host name. But if the default host has a device whose name happens to match the name of some host, you can prevent the device name from being misinterpreted as a host name by writing an extra colon at the beginning of the string: For example, **:EE:<RMS>FOO.BAR** refers to the device **EE** on the default host (assumed to use TOPS-20 syntax) rather than to the host named **EE**.

Pathnames are kept unique, like symbols, so that there is only one object with a given set of components. This is useful because a pathname object has a property list (see section 5.10, page 113) on which you can store properties describing the file or family of files that the pathname represents. The uniqueness implies that each time the same components are typed in, the program gets the same pathname object and finds there the properties it ought to find.

Note that a pathname is not necessarily the name of a specific file. Rather, it is a way to get to a file; 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 **:newest** as its version

refers to the same file as a pathname which has the appropriate number as the version. In systems with links, multiple file names, logical devices, etc., two pathnames that look quite different may really turn out to address the same file. To get from a pathname to a file requires doing a file system operation such as **open**.

When you want to store properties describing an individual file, use the pathname you get by sending **:truename** to a stream rather than the pathname you open. This avoids problems with different pathnames that refer to the same file.

To get a unique pathname object representing a family of files, send the message **:generic-pathname** to a pathname for any file in the family (see section 24.5, page 561).

24.2 Pathname Components

These are the components of a pathname. They are clarified by an example below.

host — An object that represents the file system machine on which the file resides. A host object is an instance of a flavor one of whose components is **si:basic-host**. The precise flavor varies depending on the type of file system and how the files are to be accessed.

device — Corresponds to the "device" or "file structure" concept in many host file systems.

directory — The name of a group of related files belonging to a single user or project. Corresponds to the "directory" concept in many host file systems.

name — The name of a group of files that can be thought of as conceptually the "same" file. Many host file systems have a concept of "name" which maps directly into this component.

type — Corresponds to the "filetype" or "extension" concept in many host file systems. This says what kind of file this is; such as, a Lisp source file, a QFASL file, etc.

version — Corresponds to the "version number" concept in many host file systems. This is a number that increments every time the file is modified. Some host systems do not support version numbers.

As an example, consider a Lisp program named **CONCH**. If it belongs to **GEORGE**, who uses the **FISH** machine, the host would be the host-object for the machine **FISH**, the device would probably be the default and the directory would be **GEORGE**. On this directory would be a number of files related to the **CONCH** program. The source code for this program would live in a set of files with name **CONCH**, type **LISP**, and versions **1**, **2**, **3**, etc. The compiled form of the program would live in files named **CONCH** with type **QFASL**; each would have the same version number as the source file that it came from. If the program had a documentation file, it would have type **INFO**.

Not all of the components of a pathname need to be specified. If a component of a pathname is missing, its value is **nil**. Before a file server can do anything interesting with a file, such as opening the file, all the missing components of a pathname must be filled in from defaults. But pathnames with missing components are often handed around inside the machine, since almost all pathnames typed by users do not specify all the components explicitly. The host

is not allowed to be missing from any pathname; since the behavior of a pathname is host-dependent to some extent, it has to know what its host is. All pathnames have host attributes, even if the string being parsed does not specify one explicitly.

A component of a pathname can also be the special symbol :unspecific. :unspecific means, explicitly, "this component has been specified as missing", whereas nil means that the component was not specified and should default. In merging, :unspecific counts as a specified component and is not replaced by a default. :unspecific does *not* mean "unspecified"; it is unfortunate that those two words are similar.

:unspecific is used in *generic* pathnames, which refer not to a file but to a whole family of files. The version, and usually the type, of a generic pathname are :unspecific. Another way :unspecific is used has to do with mapping of pathnames into file systems such as ITS that do not have all six components. A component that is really "not there" is :unspecific in the pathname. When a pathname is converted to a string, nil and :unspecific both cause the component not to appear in the string.

A component of a pathname can also be the special symbol :wild. This is useful only when the pathname is being used with a directory primitive such as fs:directory-list (see page 598), where it means that this pathname component matches anything. The printed representation of a pathname usually designates :wild with an asterisk; however, this is host-dependent.

What values are allowed for components of a pathname depends, in general, on the pathname's host. However, in order for pathnames to be usable in a system-independent way certain global conventions are adhered to. These conventions are stronger for the type and version than for the other components, since the type and version are actually understood by many programs, while the other components are usually just treated as something supplied by the user that only needs to be remembered.

In general, programs can interpret the components of a pathname independent of the file system; and a certain minimum set of possible values of each component are supported on all file systems. The same pathname component value may have very different representations when the pathname is made into a string, depending on the file system. This does not affect programs that operate on the components. The user, when asked to type a pathname, always uses the system-dependent string representation. This is convenient for the user who moves between using the Lisp Machine on files stored on another host and making direct use of that host. However, when the mapping between string form and components is complicated, the components may not be obvious from what you type.

The type is always a string, or one of the special symbols nil, :unspecific, and :wild. Certain hosts impose a limit on the size of string allowed, often very small. Many programs that deal with files have an idea of what type they want to use. For example, Lisp source programs are usually "LISP", compiled Lisp programs are "QFASL", etc. However, these file type conventions are host-specific, for the important reason that some hosts do not allow a string five characters long to be used as the type. Therefore, programs should use a *canonical type* rather than an actual string to specify their conventional default file types. Canonical types are described below.

For the version, it is always legitimate to use a positive fixnum, or certain special symbols. nil, :unspecific, and :wild have been explained above. The other standardly allowed symbols are :newest and :oldest. :newest refers to the largest version number that exists when reading a file, or that number plus one when writing a new file. :oldest refers to the smallest version number that exists. Some file systems may define other special version symbols, such as :installed for example, or may allow negative numbers. Some do not support versions at all. Then a pathname may still contain any of the standard version components, but it does not matter what the value is.

The device, directory, and name are more system-dependent. These can be strings (with host-dependent rules on allowed characters and length) or they can be *structured*. A structured component is a list of strings. This is used for file system features such as hierarchical directories. The system is arranged so that programs do not need to know about structured components unless they do host-dependent operations. Giving a string as a pathname component to a host that wants a structured value converts the string to the appropriate form. Giving a structured component to a host that does not understand them converts it to a string by taking the first element and ignoring the rest.

Some host file systems have features that do not fit into this pathname model. For instance, directories might be accessible as files, there might be complicated structure in the directories or names, or there might be relative directories, such as '<' in Multics. These features appear in the parsing of strings into pathnames, which is one reason why the strings are written in host-dependent syntax. Pathnames for hosts with these features are also likely to handle additional messages besides the common ones documented in this chapter, for the benefit of host-dependent programs that want to access those features. However, once your program depends on any such features, it will work only for certain file servers and not others; in general, it is a good idea to make your program work just as well no matter what file server is being used.

24.2.1 Raw Components and Interchange Components

On some host file systems it is conventional to use lower-case letters in file names, while in others upper case is customary, or possibly required. When pathname components are moved from pathnames of one file system to pathnames of another file system, it is useful to convert the case if necessary so that you get the right case convention for the latter file system as a default. This is especially useful when copying files from one file system to another.

The Lisp Machine system defines two representations for each of several pathname components (the device, directory, name and type). There is the *raw* form, which is what actually appears in the filename on the host file system, and there is the *interchange* form, which may differ in alphabetic case from the raw form. The raw form is what is stored inside the pathname object itself, but programs nearly always operate on the interchange form. The :name, :type, etc., operations return the interchange form, and the :new-name, etc., operations expect the interchange form. Additional operations :raw-name, etc., are provided for working with the raw components, but these are rarely needed.

The interchange form is defined so that it is always customarily in upper case. If upper case is customary on the host file system, then the interchange form of a component is the same as the raw form. If lower case is customary on the host file system, as on Unix, then the

interchange form has case inverted. More precisely, an all-upper-case component is changed to all-lower-case, an all-lower-case component is changed to all-upper-case, and a mixed-case component is not changed. (This is a one-to-one mapping). Thus, a Unix pathname with a name component of "foo" has an interchange-format name of "FOO", and vice versa.

For host file systems which record case when files are created but ignore case when comparing filenames, the interchange form is always upper case.

The host component is not really a name, and case is always ignored in host names, so there is no need for two forms of host component. The version component does not need them either, because it is never a string.

24.2.2 Pathname Component Operations

:host *Operation on* pathname
:device *Operation on* pathname
:directory *Operation on* pathname
:name *Operation on* pathname
:type *Operation on* pathname
:version *Operation on* pathname

These return the components of the pathname, in interchange form. The returned values can be strings, special symbols, or lists of strings in the case of structured components. The type is always a string or a symbol. The version is always a number or a symbol.

:raw-device *Operation on* pathname
:raw-directory *Operation on* pathname
:raw-name *Operation on* pathname
:raw-type *Operation on* pathname

These return the components of the pathname, in raw form.

:new-device *dev* *Operation on* pathname
:new-directory *dir* *Operation on* pathname
:new-name *name* *Operation on* pathname
:new-type *type* *Operation on* pathname
:new-version *version* *Operation on* pathname

These return a new pathname that is the same as the pathname they are sent to except that the value of one of the components has been changed. The specified component value is interpreted as being in interchange form, which means its case may be converted. The :new-device, :new-directory and :new-name operations accept a string (or a special symbol) or a list that is a structured name. If the host does not define structured components, and you specify a list, its first element is used.

:new-raw-device *dev* *Operation on* pathname
:new-raw-directory *dir* *Operation on* pathname
:new-raw-name *name* *Operation on* pathname
:new-raw-type *type* *Operation on* pathname

These return a new pathname that is the same as the pathname they are sent to except that the value of one of the components has been changed. The specified component

value is interpreted as raw.

:new-suggested-name *name* *Operation on* pathname
:new-suggested-directory *dir* *Operation on* pathname

These differ from the :new-name and :new-directory operations in that the new pathname constructed has a name or directory based on the suggestion, but not necessarily identical to it. It tries, in a system-dependent manner, to adapt the suggested name or directory to the usual customs of the file system in use.

For example, on a TOPS-20 system, these operations would convert *name* or *dir* to upper case, because while lower-case letters *may* appear in TOPS-20 pathnames, it is not customary to generate such pathnames by default.

:new-pathname &rest *options* *Operation on* pathname

This returns a new pathname that is the same as the pathname it is sent to except that the values of some of the components have been changed. *options* is a list of alternating keywords and values. The keywords all specify values of pathname components; they are :host, :device, :directory, :name, :type, and :version. Alternatively, the keywords :raw-device, :raw-directory, :raw-name and :raw-type may be used to specify a component in raw form.

Two additional keywords, :canonical-type and :original-type, allow the type field to be specified as a canonical type. See the following section for a description of canonical types. Also, the value specified for the keyword :type may be a canonical type symbol.

If an invalid component is specified, it is replaced by some valid component so that a valid pathname can be returned. You can tell whether a component is valid by specifying it in :new-pathname and seeing whether that component of the resulting pathname matches what you specified.

The operations :new-name, etc., are equivalent to :new-pathname specifying only one component to be changed; in fact, that is how those operations are implemented.

24.2.3 Canonical Types

Canonical types are a way of specifying a pathname type component using host-dependent conventions without making the program itself explicitly host dependent. For example, the function compile-file normally provides a default type of "LISP", but on VMS systems the default must be "LSP" instead, and on Unix systems it is "l". What compile-file actually does is to use a canonical type, the keyword :lisp, as the default. This keyword is given a definition as a canonical type, which specifies what it maps into on various file systems.

A single canonical type may have more than one mapping on a particular file system. For example, on TOPS-20 systems the canonical type :LISP maps into either "LISP" or "LSP". One of the possibilities is marked as "preferred"; in this case, it is "LISP". The effect of this is that either FOO.LISP or FOO.LSP would be acceptable as having canonical type :lisp, but merging yields "LISP" as the type when defaulting from :lisp.

Note that the canonical type of a pathname is not a distinct component. It is another way of describing or specifying the type component.

A canonical type must be defined before it is used.

fs:define-canonical-type *Macro*

symbol standard-mapping system-dependent-mappings...

Defines *symbol* as a canonical type. *standard-mapping* is the actual type component that it maps into (a string), with exceptions as specified by *system-dependent-mappings*. Each element of *system-dependent-mappings* (that is, each additional argument) is a list of the form

(*system-type preferred-mapping other-mappings...*)

system-type is one of the system-type keywords the :system-type operation on a host object can return, such as :unix, :tops20, and :lispm (see page 577). The argument describes how to map this canonical type on that type of file system. *preferred-map* (a string) is the preferred mapping of the canonical type, and *other-mappings* are additional strings that are accepted as matching the canonical type.

system-type may also be a list of system types. Then the argument applies to all of those types of file systems.

All of the mapping strings are in interchange form.

For example, the canonical type :lisp is defined as follows:

```
(fs:define-canonical-type :lisp "LISP"
  (:unix "L" "LISP")
  (:vms "LSP")
  ((:tops20 :tenex) "LISP" "LSP"))
```

Other canonical types defined by the system include :qfasl, :text, :press, :qwabl, :babyl, :mail, :xmail, :init, :patch-directory, :midas, :palx, :unfasl, :widths, :output, mac, tasm, doc, mss, tex, pl1 and clu. The standard mapping for each is the symbol's pname.

To match a pathname against a canonical type, use the :canonical-type operation.

:canonical-type *Operation on* **pathname**

Returns two values which describe whether and how this pathname's type component matches any canonical type.

If the type component is one of the possible mappings of some canonical type, the first value is that canonical type (the symbol). The second value is nil if the type component is the preferred mapping of the canonical type; otherwise it is the actual type component, in interchange form. The second value is called the *original type* of the pathname.

If the type component does not match a canonical type, the first value is the type component in interchange form (a string), and the second value is nil.

This operation is useful in matching a pathname against a canonical type; the first value is eq to the canonical type if the pathname matches it. The operation is also useful for transferring a type field from one file system to another while preserving canonical type; this is described below.

A new pathname may also be constructed by specifying a canonical type.

:new-canonical-type *Operation on* pathname

canonical-type &optional *original-type*

Returns a pathname different from this one in having a type component that matches *canonical-type*.

If *original-type* is a possible mapping for *canonical-type* on this pathname's host, then it is used as the type component. Otherwise, the preferred mapping for *canonical-type* is used. If *original-type* is not specified, it defaults to this pathname's type component. If it is specified as nil, the preferred mapping of the canonical type is always used. If *canonical-type* is a string rather than an actual canonical type, it is used directly as the type component, and the *original-type* does not matter.

The :new-pathname operation accepts the keywords :canonical-type and :original-type. The :new-canonical-type operation is equivalent to :new-pathname with those keywords.

Suppose you wish to copy the file named *old-pathname* to a directory named *target-directory-pathname*, possibly on another host, while preserving the name, version and canonical type. That is, if the original file has a name acceptable for a QFASL file, the new file should also. Here is how to compute the new pathname:

```
(multiple-value-bind (canonical original)
    (send old-pathname :canonical-type)
  (send target-directory-pathname :new-pathname
        :name (send old-pathname :name)
        :version (send old-pathname :version)
        :canonical-type canonical
        :original-type original))
```

Suppose that *old-pathname* is OZ:<FOO>A.LISP.5, where OZ is a TOPS-20, and the target directory is on a VMS host. Then canonical is :lisp and original is "LISP". Since "LISP" is not an acceptable mapping for :lisp on a VMS system, the resulting pathname has as its type component the preferred mapping for :lisp on VMS, namely, "LSP".

But if the target host is a Unix host, the new file's type is "LISP", since that is an acceptable (though not preferred) mapping for :lisp on Unix hosts. If you would rather that the preferred mapping always be used for the new file's type, omit the :original-type argument to the :new-pathname operation. This would result in a type component of "L" in interchange form, or "l" in raw form, in the new file's pathname.

The function compile-file actually does something cleverer than using the canonical type as a default. Doing that, and opening the resulting pathname, would look only for the preferred mapping of the canonical type. compile-file actually tries to open *each* possible mapping, trying

the preferred mapping first. Here is how it does so:

:open-canonical-default-type *Operation on* pathname *canonical-type* &rest *options*

If this pathname's type component is non-nil, the pathname is simply opened, passing the *options* to the :open operation. If the type component is nil, each mapping of *canonical-type* is tried as a type component, in the order the mappings appear in the canonical type definition. If an open succeeds, a stream is returned. The possibilities continue to be tried as long as fs:file-not-found errors happen; other errors are not handled. If all the possibilities fail, a fs:file-not-found error is signaled for the caller, with a pathname that contains the preferred mapping as its type component.

24.3 Defaults and Merging

When the user is asked to type in a pathname, it is of course unreasonable to require the user to type a complete pathname, containing all components. Instead there are defaults, so that components not specified by the user can be supplied automatically by the system. Each program that deals with pathnames typically has its own set of defaults.

The system defines an object called a *defaults alist*. Functions are provided to create one, get the default pathname out of one, merge a pathname with one, and store a pathname back into one. A defaults alist can remember more than one default pathname if defaults are being kept separately for each host; this is controlled by the variable fs:*defaults-are-per-host*. The main primitive for using defaults is the function fs:merge-pathname-defaults (see page 558).

In place of a defaults alist, you may use just a pathname. Defaulting one pathname from another is useful for cases such as a program that has an input file and an output file, and asks the user for the name of both, letting the unsupplied components of one name default from the other. Unspecified components of the output pathname come from the input pathname, except that the type should default not to the type of the input but to the appropriate default type for output from this program.

The implementation of a defaults alist is an association list of host names and default pathnames. The host name nil is special and holds the defaults for all hosts, when defaults are not per-host.

The *merging* operation takes as input a pathname, a defaults alist (or another pathname), a default type, and a default version, and returns a pathname. Basically, the missing components in the pathname are filled in from the defaults alist. However, if a name is specified but the type or version is not, then the type or version is treated specially.

Here are the merging rules in full detail.

If no host is specified, the host is taken from the defaults. If the pathname explicitly specifies a host and does not supply a device, then the the default file device for that host is used.

If the pathname specifies a device named DSK, that is replaced with the *working device* for the pathname's host, and the directory defaults to the *working directory* for the host if it is not specified. See fs:set-host-working-directory, below.

Next, if the pathname does not specify a host, device, directory, or name, that component comes from the defaults.

If the value of fs:*always-merge-type-and-version* is non-nil, the type and version are merged just like the other components.

If fs:*always-merge-type-and-version* is nil, as it normally is, the merging rules for the type and version are more complicated and depend on whether the pathname specifies a name. If the pathname doesn't specify a name, then the type and version, if not provided, come from the defaults, just like the other components. However, if the pathname does specify a name, then the type and version come from the *default-type* and *default-version* arguments to merge-pathname-defaults. If those arguments were omitted, the value of fs:*name-specified-default-type* (initially, :lisp) is used as the default type, and :newest is used as the default version.

The reason for this is that the type and version "belong to" some other filename, and are thought to be unlikely to have anything to do with the new filename you are typing in.

fs:set-host-working-directory *host pathname*

Sets the *working device* and *working directory* for *host* to those specified in *pathname*. *host* should be a host object or the name of a host. *pathname* may be a string or a pathname. The working device and working directory are used for defaulting pathnames in which the device is specified as DSK.

The editor command Meta-X Set Working Directory provides a convenient interface to this function.

The following special variables are parts of the pathname interface that are relevant to defaults.

fs:*defaults-are-per-host* *Variable*

This is a user customization option intended to be set by a user's LISPM INIT file (see section 35.8, page 800). The default value is nil, which means that each program's set of defaults contains only one default pathname. If you type in just a host name and a colon, the other components of the name default from the previous host, with appropriate translation to the new host's pathname syntax. If fs:*defaults-are-per-host* is set to t, each program's set of defaults maintains a separate default pathname for each host. If you type in just a host name and a colon, the last file that was referenced on that host is used.

fs:*always-merge-type-and-version* *Variable*

If this variable is non-nil, then the type and version are defaulted only from the pathname defaults just like the other components.

fs:*name-specified-default-type* *Variable*

If fs:*always-merge-type-and-version* is nil, then when a name is specified but not a type, the type defaults from an argument to the merging function. If that argument is not specified, this variable's value is used. It may be a string or a canonical type keyword. The value is initially :lisp.

default-pathname-defaults *Variable*

This is the default defaults alist: if the pathname primitives that need a set of defaults are not given one, they use this one. Most programs, however, should have their own defaults rather than using these.

cli:*default-pathname-defaults* *Variable*

The Common Lisp version of the default pathname defaults. The value of this variable is a pathname rather than an alist. This variable is magically (with a forwarding pointer) identified with a cell in the defaults-alist which the system really uses, so that setting this variable modifies the contents of the alist.

fs:last-file-opened *Variable*

This is the pathname of the last file that was opened. Occasionally this is useful as a default. Since some programs deal with files without notifying the user, you must not expect the user to know what the value of this symbol is. Using this symbol as a default may cause unfortunate surprises if you don't announce it first, and so such use is discouraged.

These functions are used to manipulate defaults alists directly.

fs:make-pathname-defaults

Creates a defaults alist initially containing no defaults. If you ask this empty set of defaults for its default pathname before anything has been stored into it you get the file FOO on the user's home directory on the host he logged in to.

fs:copy-pathname-defaults *defaults*

Creates a defaults alist, initially a copy of *defaults*.

fs:default-pathname &optional *defaults host default-type default-version*

This is the primitive function for getting a default pathname out of a defaults alist. Specifying the optional arguments *host*, *default-type*, and *default-version* to be non-nil forces those fields of the returned pathname to contain those values.

If fs:*defaults-are-per-host* is nil (its default value), this gets the one relevant default from the alist. If it is t, this gets the default for *host* if one is specified, otherwise for the host most recently used.

If *defaults* is not specified, the default defaults are used.

This function also has an additional optional argument *internal-p*, which is obsolete.

fs:default-host *defaults*

Returns the default host object specified by the defaults-alist *defaults*. This is the host used by pathname defaulting with the given defaults if no host is specified.

fs:set-default-pathname *pathname* &optional *defaults*
This is the primitive function for updating a set of defaults. It stores *pathname* into *defaults*. If *defaults* is not specified, the default defaults are used.

24.4 Pathname Functions

This function obtains a pathname from an object if that is possible.

pathname *object*
Converts *object* to a pathname and returns that, if possible. If *object* is a string or symbol, it is parsed. If *object* is a plausible stream, it is asked for its pathname with the :pathname operation. If *object* is a pathname, it is simply returned. Any other kind of *object* causes an error.

These functions are what programs use to parse and default file names that have been typed in or otherwise supplied by the user.

parse-namestring *thing* &optional *host defaults* &key (*start* 0) *end junk-allowed*
Is the Common Lisp function for parsing file names. It is equivalent to fs:parse-pathname except in that it takes some keyword arguments where the other function takes all positional arguments.

fs:parse-pathname *thing* &optional *host defaults* (*start* 0) *end junk-allowed*
This turns *thing*, which can be a pathname, a string, a symbol, or a Maclisp-style name list, into a pathname. Most functions that are advertised to take a pathname argument call fs:parse-pathname on it so that they can accept anything that can be turned into a pathname. If thing is itself a pathname, it is returned unchanged.

If *thing* is a string, *start* and *end* are interpreted as indices specifying a substring to parse. They are just like the second and third arguments to substring. The rest of *thing* is ignored. *start* and *end* are ignored if *thing* is not a string.

If *junk-allowed* is non-nil, parsing stops without error if the syntax is invalid, and this function returns nil. The second value is then the index of the invalid character. If parsing is successful, the second value is the index of the place at which parsing was supposed to stop (*end*, or the end of *thing*). If *junk-allowed* is nil, invalid syntax signals an error.

This function does *not* do defaulting, even though it has an argument named *defaults*; it only does parsing. The *host* and *defaults* arguments are there because in order to parse a string into a pathname, it is necessary to know what host it is for so that it can be parsed with the file name syntax peculiar to that host. If *thing* does not contain a manifest host name, then if *host* is non-nil, it is the host name to use, as a string. If *thing* is a string, a manifest host name may be at the beginning or the end, and consists of the name of a host followed by a colon. If *host* is nil then the host name is obtained from the default pathname in *defaults*. If *defaults* is not supplied, the default defaults (*default-pathname-defaults*) are used.

Note that if *host* is specified, and *thing* contains a host name, an error is signaled if they are not the same host.

fs:pathname-parse-error (fs:pathname-error error) *Condition*
This condition is signaled when fs:parse-pathname finds a syntax error in the string it is given.

fs:parse-pathname sets up a nonlocal proceed type :new-pathname for this condition. The proceed type expects one argument, a pathname, which is returned from fs:parse-pathname.

fs:merge-pathname-defaults *pathname* &optional *defaults default-type default-version*
Fills in unspecified components of *pathname* from the defaults and returns a new pathname. This is the function that most programs should call to process a file name supplied by the user. *pathname* can be a pathname, a string, a symbol, or a Maclisp namelist. The returned value is always a pathname. The merging rules are documented on page 554.

If *defaults* is a pathname, rather than a defaults alist, then the defaults are taken from its components. This is how you merge two pathnames. (In Maclisp that operation is called mergef.)

defaults defaults to the value of *default-pathname-defaults* if unsupplied. *default-type* defaults to the value of fs:*name-specified-default-type*. *default-version* defaults to :newest.

merge-pathnames *pathname* &optional *defaults* (*default-version* :newest)
Is the Common Lisp function for pathname defaulting. It does only some of the things that fs:merge-pathname-defaults does. It merges defaults from *defaults* (which defaults to the value of *default-pathname-defaults*) into *pathname* to get a new pathname, which is returned. *pathname* can be a string (or symbol); then it is parsed and the result is defaulted. *default-version* is used as the version when pathname has a name but no version.

fs:merge-and-set-pathname-defaults *pathname* &optional *defaults default-type default-version*
This is the same as fs:merge-pathname-defaults except that after it is done the defaults-list *defaults* is modified so that the merged pathname is the new default. This is handy for programs that have sticky defaults, which means that the default for each command is the last filename used. (If *defaults* is a pathname rather than a defaults alist, then no storing back is done.) The optional arguments default the same way as in fs:merge-pathname-defaults.

These functions convert a pathname into a namestring for all or some of the pathname's components.

namestring *pathname*
Returns a string containing the printed form of *pathname*, as you would type it in. This uses the :string-for-printing operation.

file-namestring *pathname*
Returns a string showing just the name, type and version of *pathname*. This uses the :string-for-dired operation.

directory-namestring *pathname*
Returns a string showing just the device and directory of *pathname*. This uses the :string-for-directory operation.

enough-namestring *pathname* &optional *defaults*
Returns a string showing just the components of *pathname* which would not be obtained by defaulting from *defaults*. This is the shortest string that would suffice to specify pathname, given those defaults. It is made by using the :string-for-printing operation on a modified pathname.

This function yields a pathname given its components.

make-pathname &key (*defaults* t) *host device raw-device directory raw-directory name raw-name type raw-type version canonical-type original-type*
Returns a pathname whose components are as specified.

If *defaults* is a pathname or a defaults-alist, any components not explicitly specified default from it. If *defaults* is t (which is the default), then unspecified components default to nil, except for the host (since every pathname must have a specific host), which defaults based on *default-pathname-defaults*.

These functions give the components of a pathname.

pathname-host *pathname*
Returns the host component of *pathname*.

pathname-device *pathname*
pathname-directory *pathname*
pathname-name *pathname*
pathname-type *pathname*
pathname-version *pathname*
Likewise, for the other components

These functions return useful information.

fs:user-homedir &optional *host reset-p* (*user* user-id) *force-p*
user-homedir-pathname &optional *host reset-p* (*user* user-id) *force-p*
Returns the pathname of the *user*'s home directory on *host*. These default to the logged in user and the host logged in to. Home directory is a somewhat system-dependent concept, but from the point of view of the Lisp Machine it is the directory where the user keeps personal files such as init files and mail.

This function returns a pathname without any name, type, or version component (those components are all nil).

If *reset-p* is specified non-nil, the machine the user is logged in to is changed to be *host*.

The synonym user-homedir-pathname is from Common Lisp.

init-file-pathname *program-name* &optional *host*
Returns the pathname of the logged-in user's init file for the program *program-name*, on the *host*, which defaults to the host the user logged in to. Programs that load init files containing user customizations call this function to find where to look for the file, so that they need not know the separate init file name conventions of each host operating system. The *program-name* "LISPM" is used by the login function.

These functions are useful for poking around.

fs:describe-pathname *pathname*
If *pathname* is a pathname object, this describes it, showing you its properties (if any) and information about files with that name that have been loaded into the machine. If *pathname* is a string, this describes all interned pathnames that match that string, ignoring components not specified in the string. One thing this is useful for is finding the directory of a file whose name you remember. Giving describe (see page 791) a pathname object invokes this function.

fs:pathname-plist *pathname*
Parses and defaults *pathname*, then returns the list of properties of that pathname.

fs:*pathname-hash-table* *Variable*
This is the hash table in which pathname objects are interned. You can find all pathnames ever constructed by applying the function maphash to this hash table.

24.5 Generic Pathnames

A generic pathname stands for a whole family of files. The property list of a generic pathname is used to remember information about the family, some of which (such as the package) comes from the -*- line (see section 25.5, page 594) of a source file in the family. Several types of files with that name, in that directory, belong together. They are different members of the same family; for example, they may be source code and compiled code. However, there may be several other types of files that form a logically distinct group even though they have this same name; TEXT and PRESS for example. The exact mapping is done on a per host basis since it can sometimes be affected by host naming conventions.

The generic pathname of pathname *p* usually has the same host, device, directory, and name as *p* does. However, it has a version of :unspecific. The type of the generic pathname is obtained by sending a :generic-base-type *type-of-p* message to the host of *p*. The default response to this message is to return the associated type from fs:*generic-base-type-alist* if there is one, else *type-of-p*. Both the argument and the value are either strings, in interchange form, or canonical type symbols.

However, the ITS file system presents special problems. One cannot distinguish multiple generic base types in this same way since the type component does not exist as such; it is derived from the second filename, which unfortunately is also sometimes used as a version number. Thus, on ITS, the type of a generic pathname is always :unspecific if there is any association for the type of the pathname on fs:*generic-base-type-alist*.

Since generic pathnames are primarily useful for storing properties, it is important that they be as standardized and conceptualized as possible. For this reason, generic pathnames are defined to be backtranslated, i.e. the generic pathname of a pathname that is (or could be) the result of a logical host translation has the host and directory of the logical pathname. For example, the generic pathname of OZ:<L.WINDOW>;STREAM LISP would be SYS:WINDOW;STREAM U U if OZ is the system host.

All version numbers of a particular pathname share the same identical generic pathname. If the values of particular properties have changed between versions, it is possible for confusion to result. One way to deal with this problem is to have the property be a list associating version number with the actual desired property. Then it is relatively easy to determine which versions have which values for the property in question and select one appropriately. But in the applications for which generic pathnames are typically used, this is not necessary.

The :generic-pathname operation on a pathname returns its corresponding generic pathname. See page 563. The :source-pathname operation on a pathname returns the actual or probable pathname of the corresponding source file (with :newest as the version). See page 563.

fs:*generic-base-type-alist* *Variable*

This is an association list of the file types and the type of the generic pathname used for the group of which that file type is a part. Constructing a generic pathname replaces the file type with the association from this list, if there is one (except that ITS hosts always replace with :unspecific). File types not in this list are really part of the name in some sense. The initial list is

```
((:text . :text) ("DOC" . :text)
 (:press . :text) ("XGP" . :text)
 (:lisp . :unspecific) (:qfasl . :unspecific)
 (nil . :unspecific))
```

The association of :lisp and :unspecific is unfortunately made necessary by the problems of ITS mentioned previously. This way makes the generic pathnames of logically mapped LISP files identical no matter whether the logical host is mapped to an ITS host or not.

The first entry in the list with a particular cdr is the entry for the type that source files have. Note how the first element whose cdr is :unspecific is the one for :lisp. This is how the :source-pathname operation knows what to do, by default.

Some users may need to add to this list.

The system records certain properties on generic pathnames automatically.

:warnings This property is used to record compilation and other warnings for the file.

:definitions This property records all the functions and other things defined in the file. The value has one element for each package into which the file has been loaded; the element's car is the package itself and the cdr is a list of definitions made.

Each definition is a cons whose car is the symbol or function spec defined and whose cdr is the type of definition (usually one of the symbols defun, defvar, defflavor and defstruct).

:systems This property's value is a list of the names of all the systems (defined with defsystem, see page 660) of which this is a source file.

:file-id-package-alist
This property records what version of the file was most recently loaded. In case the file has been loaded into more than one package, as is sometimes necessary, the loaded version is remembered for each package separately. This is how make-system tells whether a file needs to be reloaded. The value is a list with an element for each package that the file has been loaded into; the elements look like

```
(package file-information)
```

package is the package object itself; *file-information* is the value returned by the :info operation on a file stream, and is usually a cons whose car is the truename (a pathname) and whose cdr is the file creation date (a universal time number).

Some additional properties are put on the generic pathname by reading the attribute list of the file (see page 597). It is not completely clear that this is the right place to store these properties, so it may change in the future. Any property name can appear in the attributes list and get onto the generic pathname; the standard ones are described in section 25.5, page 594.

24.6 Pathname Operations

This section documents the operations a user may send to a pathname object. Pathnames handle some additional operations that are only intended to be sent by the file system itself, and therefore are not documented here. Someone who wants to add a new host to the system would need to understand those internal operations.

The operations on pathnames that actually operate on files are documented in section 25.4, page 592. Certain pathname flavors, for specific kinds of hosts, allow additional special purpose operations. These are documented in section 24.7, page 568 in the section on the specific host type.

:generic-pathname *Operation on* pathname

Returns the generic pathname for the family of files of which this pathname is a member. See section 24.5, page 561 for documentation on generic pathnames.

:source-pathname *Operation on* pathname

Returns the pathname for the source file in the family of files to which this pathname belongs. The returned pathname has :newest as its version. If the file has been loaded in some fashion into the Lisp environment, then the pathname type is that which the user actually used. Otherwise, the conventional file type for source files is determined from the generic pathname.

:primary-device *Operation on* pathname

Returns the default device name for the pathname's host. This is used in generating the initial default pathname for a host.

Operations dealing with wildcards.

The character * in a namestring is a *wildcard*. It means that the pathname is a really a pattern which specifies a set of possible filenames rather than a single filename. The matches any sequence of characters within a single component of the name. Thus, the component FOO* would match FOO, FOOBAR, FOOT, or any other component starting with FOO.

Any component of a pathname can contain wildcards except the host; wild hosts are not allowed because a known host is required in order to know what flavor the pathname should be. If a pathname component is written in the namestring as just *, the actual component of the pathname instance is the keyword :wild. Components which contain wildcards but are not simply a single wildcard are represented in ways subject to change.

Pathnames whose components contain wildcards are called *wild* pathnames. Wild pathnames useful in functions such as delete-file for requesting the deletion of many files at once. Less obviously but more fundamentally, wild pathnames are required for most use of the function fs:directory-list; an entire directory's contents are obtained by specifying a pathname whose name, type and version components are :wild.

:wild-p *Operation on* pathname

Returns non-nil if this pathname contains any sort of wildcards. If the value is not nil, it is a keyword, one of :device, :directory, :name, :type and :version, and it identifies the 'first' component which is wild.

:device-wild-p *Operation on* pathname

t if this pathname's device contains any sort of wildcards.

:directory-wild-p *Operation on* pathname
:name-wild-p *Operation on* pathname
:type-wild-p *Operation on* pathname
:version-wild-p *Operation on* pathname

Similar, for the other components that can be wild. (The host cannot ever be wild.)

:pathname-match *Operation on* pathname

candidate-pathname &optional (*match-host-p* t)

Returns t if candidate-pathname matches the pathname on which the operation is invoked (called, in this context, the *pattern pathname*). If the pattern pathname contains no wildcards, the pathnames match only if they are identical. This operation is intended in cases where wildcards are expected.

Wildcard matching is done individually by component; the operation returns t only if each component matches. Within each component, an occurrence of * in pattern pathname's component can match any sequence of characters in *candidate-pathname*'s component. Other characters, except for host-specific wildcards, must match exactly. :wild as a component of the pattern pathname matches any component that *candidate-pathname* may have.

Note that if a component of the pattern pathname is nil, *candidate-pathname*'s component must be nil also to match it. Most user programs that read pathnames and use them as patterns default unspecified components to :wild first.

Examples:

```
(defvar pattern)
(defun test (str)
  (send pattern
        :pathname-match
        (parse-namestring str)))

(setq pattern
      (parse-namestring "OZ:*:<F*O>*.TEXT.*"))

(test "OZ:<FOO>A.TEXT") => t
(test "OZ:<FO>HAHA.TEXT.3") => t
(test "OZ:<FPPO>HAHA.TEXT.*") => t
(test "OZ:<FOX>LOSE.TEXT") => nil

(setq pattern
      (parse-namestring "OZ:*:<*>A.TEXT*.5"))

(test "OZ:<FOO>A.TEXT.5") => t
(test "OZ:<FOO>A.TEXTTTT.5") => t
(test "OZ:<FOO>A.TEXT") => nil
```

If *match-host-p* is nil, then the host components of the two pathnames are not tested. The result then depends only on the other components.

:translate-wild-pathname *Operation on* pathname

target-pattern starting-data &optional *reversible*

Returns a pathname corresponding to *starting-data* under the mapping defined by the wild pathnames *source-pattern*, which is the pathname this operation is invoked on, and *target-pattern*, the argument. It is expected that *starting-data* would match the source pattern under the :pathname-match operation.

:translate-wild-pathname is used by functions such as copy-file which use one wild pathname to specify a set of files and a second wild pathname to specify a corresponding filename for each file in the set. The first wild pathname would be used as the source-pattern and the second, specifying the name to copy each file to, would be passed as the *target-pattern* pathname.

Each component of the result is computed individually from the corresponding components of *starting-data* and the pattern pathnames, using the following rules:

1) If *target-pattern*'s component is ;wild, then the result component is taken from *starting-data*.

2) Otherwise, each non-wild character in *target-pattern*'s component is taken literally into the result. Each wild character in *target-pattern*'s component is paired with a wild character in *source-pattern*'s component, and thereby with the portion of *starting-data*'s component which that matched. This portion of *starting-data* appears in the result in place of the wild target character.

Example:

```
(setq source (fs:parse-pathname "OZ:PS:<FOO>A*B*.*.*"))
(setq target (fs:parse-pathname "OZ:SS:<*>*LOSE*.*B.*"))

(send source :translate-wild-pathname target
      (fs:parse-pathname "OZ:PS:<FOO>ALIBI.LISP.3"))
 => the pathname OZ:SS:<FOO>LILOSEI.LISPB.3
```

It is easiest to understand the mapping as being done in interchange case: the interchange components of the arguments are used and the results specify the interchange components of the value.

The type component is slightly special; if the *target-pattern* type is :wild, the canonical type of *starting-data* is taken and then interpreted according to the mappings of the target host. Example:

```
(setq source (fs:parse-pathname "OZ:PS:<FOO>A*.*.*"))
(setq target (fs:parse-pathname "U://usr//foo//b*.*"))

(send source :translate-wild-pathname target
      (fs:parse-pathname "OZ:PS:<FOO>ALL.LISP"))
 => the pathname U:/usr/foo/b11.1
```

If *reversible* is non-nil, rule 1 is not used; rule 2 controls all mapping. This mode is used by logical pathname translation. It makes a difference when the target pattern component is :wild and the source pattern component contains wildcards but is not simply :wild. For example, with source and target pattern components BIG* and *, and starting data BIGGER, the result is ordinarily BIGGER by rule 1, but with reversible translation the result is GER.

Operations to get a path name string out of a pathname object:

:string-for-printing *Operation on* pathname

Returns a string that is the printed representation of the path name. This is the same as what you get if you princ the pathname or take string of it.

:string-for-wholine *length* *Operation on* pathname

Returns a string like the :string-for-printing, but designed to fit in *length* characters. *length* is a suggestion; the actual returned string may be shorter or longer than that. However, the who-line updater truncates the value to that length if it is longer.

:string-for-editor *Operation on* pathname

Returns a string that is the pathname with its components rearranged so that the name is first. The editor uses this form to name its buffers.

:string-for-dired *Operation on* pathname

Returns a string to be used by the directory editor. The string contains only the name, type, and version.

:string-for-directory *Operation on* pathname

Returns a string that contains only the device and directory of the pathname. It identifies one directory among all directories on the host.

:string-for-host *Operation on* pathname

Returns a string that is the pathname the way the host file system likes to see it.

Operations to move around through a hierarchy of directories:

:pathname-as-directory *Operation on* pathname

Assuming that the file described by the pathname is a directory, return another pathname specifying that *as* a directory. Thus, if sent to a pathname OZ:<RMS>FOO.DIRECTORY, it would return the pathname OZ:<RMS.FOO>. The name, type and version of the returned pathname are :unspecific.

:directory-pathname-as-file *Operation on* pathname

This is the inverse of the preceding operation. It returns a pathname specifying as a file the directory of the original pathname. The name, type and version of the original pathname are ignored.

The special symbol :root can be used as the directory component of a pathname on file systems that have a root directory.

Operations to manipulate the property list of a pathname:

:get *property-name* &optional *default-value* *Operation on* pathname
:getl *list-of-property-names* *Operation on* pathname
:putprop *value property-name* *Operation on* pathname
:remprop *property-name* *Operation on* pathname
:plist *Operation on* pathname

These manipulate the pathname's property list, and are used if you call the property list functions of the same names (see page 114) giving the pathname as the first argument. Please read the paragraph on page 546 explaining the care you must take in using property lists of pathnames.

24.7 Host File Systems Supported

This section lists the host file systems supported, gives an example of the pathname syntax for each system, and discusses any special idiosyncracies. More host types may be added in the future.

24.7.1 ITS

An ITS pathname looks like "*host*: *device*: *dir*; *name type-or-version*". The primary device is DSK: but other devices such as ML:, ARC:, DVR:, or PTR: may be used.

ITS does not exactly fit the virtual file system model, in that a file name has two components (FN1 and FN2) rather than three (name, type, and version). Consequently to map any virtual pathname into an ITS filename, it is necessary to decide whether the FN2 is the type or the version. The rule is that usually the type goes in the FN2 and the version is ignored; however, certain types (LISP and TEXT) are ignored and instead the version goes in the FN2. Also if the type is :unspecific the FN2 is the version.

Given an ITS filename, it is converted into a pathname by making the FN2 the version if it is '<', '>', or a number. Otherwise the FN2 becomes the type. ITS pathnames allow the special version symbols :oldest and :newest, which correspond to '<' and '>' respectively.

In every ITS pathname either the version or the type is :unspecific or nil; sometimes both are. When you create a new ITS pathname, if you specify only the version or only the type, the one not specified becomes :unspecific. If both are specified, the version is :unspecific unless the type is a normally-ignored type (such as LISP) in which case the version is :newest and the type is :unspecific so that numeric FN2's are found.

Each component of an ITS pathname is mapped to upper case and truncated to six characters.

Special characters (space, colon, and semicolon) in a component of an ITS pathname can be quoted by prefixing them with right horseshoe (⊃) or equivalence sign (≡). Right horseshoe is the same character code in the Lisp Machine character set as control-Q in the ITS character set.

An ITS pathname can have a structured name, which is a list of two strings, the FN1 and the FN2. In this case there is neither a type nor a version.

An ITS pathname with an FN2 but no FN1 (i.e. a type and/or version but no name) is represented with the placeholder FN1 '⊕', because ITS pathname syntax provides no way to write an FN2 without an FN1 before it.

The ITS init file naming convention is "*homedir*; *user program*".

fs:*its-uninteresting-types* *Variable*

The ITS file system does not have separate file types and version numbers; both components are stored in the "FN2". This variable is a list of the file types that are "not important"; files with these types use the FN2 for a version number. Files with other types use the FN2 for the type and do not have a version number. The initial list is

```
("LISP" "TEXT" nil :unspecific)
```
Some users may need to add to this list.

:fn1 *Operation on* its-pathname
:fn2 *Operation on* its-pathname
These two operations return a string that is the FN1 or FN2 host-dependent component of the pathname.

:type-and-version *Operation on* pathname
:new-type-and-version *new-type new-version* *Operation on* pathname
These two operations provide a way of pretending that ITS pathnames can have both a type and a version. They use the first three characters of the FN2 to store a type and the last three to store a version number.

On an ITS-pathname, :type-and-version returns the type and version thus extracted (not the same as the type and version of the pathname). :new-type-and-version returns a new pathname constructed from the specified new type and new version.

On any other type of pathname, these operations simply return or set both the type component and the version component.

24.7.2 TOPS-20 (Twenex), Tenex, and VMS.

A pathname on TOPS-20 (better known as Twenex) looks like
host: *device*: <*directory*> *name*. *type*. *version*
The primary device is PS:.

TOPS-20 pathnames are mapped to upper case. Special characters (including lower-case letters) are quoted with the circle-cross (⊗) character, which has the same character code in the Lisp Machine character set as Control-V, the standard Twenex quoting character, in the ASCII character set.

If you specify a period after the name, but nothing after that, then the type is :unspecific, which translates into an empty extension on the TOPS-20 system. If you omit the period, you have allowed the type to be defaulted.

TOPS-20 pathnames allow the special version symbols :oldest and :newest. In the string form of a pathname, these are expressed as '.-2', and as an omitted version.

The directory component of a TOPS-20 pathname may be structured. The directory <FOO.BAR> is represented as the list ("FOO" "BAR").

The characters * and % are wildcards that match any sequence of characters and any single character (within one pathname component), respectively. To specify a filename that actually contains a * or % character, quote the character with ⊗. When a component is specified with just a single *, the symbol :wild appears in the pathname object.

The TOPS-20 init file naming convention is "<*user*>*program*.INIT".

When there is an attempt to display a TOPS-20 file name in the who-line and there isn't enough room to show the entire name, the name is truncated and followed by a center-dot character to indicate that there is more to the name than can be displayed.

Tenex pathnames are almost the same as TOPS-20 pathnames, except that the version is preceeded by a semi-colon instead of a period, the default device is DSK instead of PS, and the quoting requirements are slightly different.

VMS pathnames are basically like TOPS-20 pathnames, with a few complexities. The primary device is USRD$.

First of all, only alphanumeric characters are allowed in filenames (though $ and underscore can appear in device names).

Secondly, a version number is preceded by ';' rather than by '.'.

Thirdly, file types (called "extensions" in VMS terminology) are limited to three characters. Each of the system's canonical types has a special mapping for VMS pathnames, which is three characters long:

```
:lisp → LSP     :text → TXT     :qfasl → QFS     :midas → MID
:press → PRS    :widths → WID   :patch-directory → PDR
:qwabl → QWB    :babyl → BAB    :mail → MAI      :xmail → XML
:init → INI     :unfasl → UNF   :output → OUT
```

24.7.3 Unix and Multics Pathnames

A Unix pathname is a sequence of directory or file names separated by slashes. The last name is the filename; preceding ones are directory names (but directories are files anyway). There are no devices or versions. Alphabetic case is significant in Unix pathnames, no case conversion is normally done, and lower case is the default. Therefore, components of solid upper or lower case are inverted in case when going between interchange form and raw form. (What the user types in a pathname string is the raw form.)

Unix allows you to specify a pathname relative to your default directory by using just a filename, or starting with the first subdirectory name; you can specify it starting from the root directory by starting with a slash. In addition, you can start with '..' as a directory name one or more times, to refer upward in the hierarchy from the default directory.

Unix pathnames on the Lisp Machine provide all these features too, but the canonicalization to a simple descending list of directory names starting from the root is done on the Lisp Machine itself when you merge the specified pathname with the defaults.

If a pathname string starts with a slash, the pathname object that results from parsing it is called "absolute". Otherwise the pathname object is called "relative".

In an absolute pathname object, the directory component is either a symbol (nil, :unspecific or :root), a string, or a list of strings. A single string is used when there is only one level of directory in the pathname.

A relative pathname has a directory that is a list of the symbol :relative followed by some strings. When the pathname is merged with defaults, the strings in the list are appended to the strings in the default directory. The result of merging is always an absolute pathname.

In a relative pathname's string form, the string ".." can be used as a directory name. It is translated to the symbol :up when the string is parsed. That symbol is processed when the relative pathname is merged with the defaults.

Restrictions on the length of Unix pathnames require abbreviations for the standard Zetalisp pathname types, just as for VMS. On Unix the preferred mappings of all canonical types are one or two characters long. We give here the mappings in raw form; they are actually specified in interchange form.

```
:lisp → l       :text → tx      :qfasl → qf      :midas → md
:press → pr     :widths → wd    :patch-directory → pd
:qwabl → qw     :babyl → bb     :mail → ma       :xmail → xm
:init → in      :unfasl → uf    :output → ot
```

The Multics file system is much like the Unix one; there are absolute and relative pathnames, absolute ones start with a directory delimiter, and there are no devices or versions. Alphabetic case is significant.

There are differences in details. Directory names are terminated, and absolute pathnames begun, with the character '>'. The containing directory is referred to by the character '<', which is complete in itself. It does not require a delimiter. Thus, <<FOO>BAR refers to subdirectory FOO, file BAR in the superdirectory of the superdirectory of the default directory.

The limits on filename sizes are very large, so the system canonical types all use their standard mappings. Since the mappings are specified as upper case, and then interpreted as being in interchange form, the actual file names on Multics contain lower case.

24.7.4 Lisp Machine File Systems

There are two file systems that run in the MIT Lisp Machine system. They have different pathname syntax. Both can be accessed either remotely like any other file server, or locally.

The Local-File system uses host name LM for the machine you are on. A Local-File system on another machine can be accessed using the name of that machine as a host name, provided that machine is known as a file server.

The remainder of the pathname for the Local-File system looks like "*directory*; *name.type # version*". There is no restriction on the length of names; letters are converted to upper case. Subdirectories are allowed and are specified by putting periods between the directory components, as in RMS.SUBDIR;.

The TOPS-20 pathname syntax is also accepted. In addition, if the flag fs:*lmfs-use-twenex-syntax* is non-nil, Local-File pathnames print out using TOPS-20 syntax. Note that since the printed representation of a pathname is cached, changing this flag's value does not change the printing of pathnames with existing representations.

The Local-File system on the filecomputer at MIT has the host name FS.

The LMFILE system is primarily for use as a file server, unless you have 512k of memory. At MIT it runs on the filecomputer and is accessed remotely with host name FC.

The remainder of an LMFILE pathname looks like "*directory*; *name type* # *version*". However, the directory and name can be composed of any number of subnames, separated by backslashes. This is how subdirectories are specified. FOO;BAR\X refers to the same file as FOO\BAR;X, but the two ways of specifying the file have different consequences in defaulting, getting directory listings, etc.

Case is significant in LMFILE pathnames; however, when you open a file, the LMFILE system ignores the case when it matches your pathname against the existing files. As a result, the case you use matters when you create or rename a file, and appears in directory listings, but it is ignored when you refer to an existing file, and you cannot have two files whose names differ only in case. When components are accessed in interchange form, they are always converted to upper case.

24.7.5 Logical Pathnames

There is another kind of pathname that doesn't correspond to any particular file server. It is called a *logical* pathname, and its host is called a logical host. Every logical pathname can be translated into a corresponding *physical* pathname because each logical host records a corresponding actual ("physical") host and rules for translating the other components of the pathname.

The reason for having logical pathnames is to make it easy to keep bodies of software on more than one file system. An important example is the body of software that constitutes the Lisp Machine system. Every site has a copy of all of the sources of the programs that are loaded into the initial Lisp environment. Some sites may store the sources on an ITS file system, while others may store them on a TOPS-20. However, system software (including make-system) wishes to be able to find a particular file independent of the name of the host a particular site stores it on, or even the kind of host it is. This is done by means of the logical host SYS; all pathnames for system files are actually logical pathnames with host SYS. At each site, SYS is defined as a logical host, but translations are different at each site. For example, at MIT the source files are stored on the TOPS-20 system named OZ, so MIT's site file says that SYS should translate to the host OZ.

Each logical host, such as SYS, has a list of translations, each of which says how to map certain pathnames for that host into pathnames for the corresponding physical host. To translate a logical pathname, the system tests each of the logical host's translations, in sequence, to see if it is applicable. (If none is applicable, an error is signaled.) A translation consists of a pair of pathnames or namestrings, typically containing wildcards. Unspecified components in them default

to :wild. The *from*-pathname of the translation is used to match against the pathname to be translated; if it matches, the corresponding *to*-pathname is used to construct the translation, filling in its wild fields from the pathname being translated as in the :translate-wild-pathname operation (page 565).

Most commonly the translations contain pathnames that have only directories specified, everything else wild. Then the other components are unchanged by translation.

If the files accessed through the logical host are moved, the translations can be changed so that the same logical pathnames refer to the same files on their new physical host via physical pathnames changed to fit the restrictions and the conventions of the new physical host.

Each translation is specified as a list of two strings. The strings are parsed into pathnames and any unspecified components are defaulted to :wild. The first string of the pair is the source pattern; it is parsed with logical pathname syntax. The second string is the target pattern, and it is parsed with the pathname syntax for the specified physical host.

For example, suppose that logical host FOO maps to physical host BAR, a Tops-20, and has the following list of translations:

```
(("BACK;" "PS:<FOO.BACK>")
 ("FRONT;* QFASL" "SS:<FOO.QFASL>*.QFASL")
 ("FRONT;" "PS:<FOO.FRONT>"))
```

Then all pathnames with host FOO and directory BACK translate to host BAR, device PS and directory <FOO.BACK> with name, type and version unchanged. All pathnames with host FOO, directory FRONT and type QFASL translate to host BAR, device SS, directory <FOO.QFASL> and type QFASL, with name and version unchanged. All other pathnames with host FOO and directory FRONT map to host BAR, device PS and directory <FOO.FRONT>, with name, type and version unchanged. Note that the first translation whose pattern matches a given pathname is the one that is used.

Another site might define FOO's to map to a Unix host QUUX, with the following translation list:

```
(("BACK;" "//nd//foo//back//")
 ("FRONT;" "//nd//foo//front//"))
```

This site apparently does not see a need to store the QFASL files in a separate directory. Note that the slashes are duplicated to quote them for Lisp; the actual namestrings contain single slashes as is usual with Unix.

If the last translation's source pattern is entirely wild, it applies to any pathname not so far handled. Example:

```
(("BACK;" "//nd//foo//back//")
 ("" "//nd//foo1//*//"))
```

Physical pathnames can also be *back-translated* into the corresponding logical pathname. This is the inverse transformation of ordinary translation. It is necessary to specify which logical host to back translate for, as it may be that the same physical pathname could be the translation of different logical pathnames on different hosts. Use the :back-translated-pathname operation, below.

fs:add-logical-pathname-host *logical-host physical-host translations*
fs:set-logical-pathname-host *logical-host* &key *physical-host translations*

Both create a new logical host named *logical-host*. Its corresponding physical host (that is, the host to which it should forward most operations) is *physical-host*. *logical-host* and *physical-host* should both be strings. *translations* should be a list of translation specifications, as described above. The two functions differ only in that one accepts positional arguments and the other accepts keyword arguments. Example:

```
(add-logical-pathname-host "MUSIC" "MUSIC-10-A"
       '(("MELODY;" "SS:<MELODY>")
         ("DOC;" "PS:<MUSIC-DOCUMENTATION>")))
```

This creates a new logical host called MUSIC. An attempt to open the file MUSIC:DOC;MANUAL TEXT 2 will be re-directed to the file MUSIC-10-A:PS:<MUSIC-DOCUMENTATION>MANUAL.TEXT.2 (assuming that the host MUSIC-10-A is a TOPS-20 system).

fs:make-logical-pathname-host *name*

Requests that the definition of logical host *name* be loaded from a standard place in the file system: namely, the file SYS: SITE; *name* TRANSLATIONS. This file is loaded immediately with load, in the fs package. It should contain code to create the logical host; normally, a call to fs:set-logical-pathname-host or fs:add-logical-pathname-host, above.

The same file is automatically reloaded, if it has been changed, at appropriate times: by load-patches, and whenever site information is updated.

:translated-pathname *Operation on* fs:logical-pathname

Converts a logical pathname to a physical pathname. It returns the translated pathname of this instance, a pathname whose host component is the physical host that corresponds to this instance's logical host.

If this operation is applied to a physical pathname, it simply returns that pathname unchanged.

:back-translated-pathname *pathname* *Operation on* fs:logical-pathname

Converts a physical pathname to a logical pathname. *pathname* should be a pathname whose host is the physical host corresponding to this instance's logical host. This returns a pathname whose host is the logical host and whose translation is *pathname*. If *pathname* is not the translation of any logical pathname on this instance's host, nil is returned.

Here is an example of how this would be used in connection with truenames. Given a stream that was obtained by opening a logical pathname,

```
(send stream :pathname)
```

returns the logical pathname that was opened.

```
(send stream :truename)
```

returns the true name of the file that is open, which of course is a pathname on the physical host. To get this in the form of a logical pathname, one would do

```
(send (send stream :pathname)
      :back-translated-pathname
      (send stream :truename))
```

If this operation is applied to a physical pathname, it simply returns its argument. Thus the above example works no matter what kind of pathname was opened to create the stream.

fs:unknown-logical-pathname-translation (fs:pathname-error error) *Condition*
This is signaled when a logical pathname has no translation. The condition instance supports the :logical-pathname operation, which returns the pathname that was untranslatable.

The proceed type :define-directory is supported. It expects a single argument, a pathname or a string to be parsed into one. This defines the target pattern for a translation whose source pattern is the directory from the untranslatable pathname (and all else wild). Such a translation is added to the logical host, making it possible to translate the pathname.

A logical pathname looks like "*host*: *directory*; *name type version*". There is no way to specify a device; parsing a logical pathname always returns a pathname whose device component is :unspecific. This is because devices don't have any meaning in logical pathnames.

The equivalence-sign character (≡) can be used for quoting special characters such as spaces and semicolons. The double-arrow character ('↔') can be used as a place-holder for components that are nil, and the up-horseshoe ('∪') indicates :unspecific (generic pathnames typically have :unspecific as the type and the version). All letters are mapped to upper case unless quoted. The :newest, :oldest, and :wild values for versions are written as '>', '<', and '*' respectively.

There isn't any init file naming convention for logical hosts; you can't log into them. The :string-for-host, :string-for-wholine, :string-for-dired, and :string-for-editor messages are all passed on to the translated pathname, but the :string-for-printing is handled by the fs:logical-pathname flavor itself and shows the logical name.

24.7.6 Editor Buffer Pathnames

The hosts ED, ED-BUFFER and ED-FILE are used in pathnames which refer to buffers in the editor. If you open such a pathname, you get a stream that reads or writes the contents of an editor buffer. The three host names differ only in the syntax of the pathname, and in how it is interpreted.

The host ED is followed by an abbreviation that should complete to the name of an existing editor buffer. For example, the pathname ED:FOO could refer to the buffer FOO.LISP PS:<ME> OZ:.

The host ED-BUFFER is followed by an exact buffer name. If there is no buffer with that name, one is created. This is most useful for creating a buffer.

The host ED-FILE is followed by an arbitrary pathname, including a host name. An ED-FILE pathname refers to a buffer visiting that file. If necessary, the file is read into the editor. For example, ED-FILE: OZ: PS:<ME>FOO.LISP would refer to the same buffer as ED: FOO. The current default defaults are used in processing the pathname that follows ED-FILE, when the pathname is parsed.

24.8 Hosts

Each host known to the Lisp Machine is represented by a flavor instance known as a host object. The host object records such things as the name(s) of the host, its operating system type, and its network address(es). Host objects print like #⊂FS:TOPS20-CHAOS-HOST "MIT-OZ"⊃, so they can be read back in.

Not all hosts support file access. Those that do support it appear on the list fs:*pathname-host-list* and can be the host component of pathnames. A host object is also used as an argument when you make a Chaosnet connection for any purpose.

The hosts that you can use for making network connections appear in the value of si:host-alist. Most of the hosts you can use for pathnames are among these; but some, such as logical hosts, are not.

24.8.1 Parsing Hostnames

si:parse-host *namestring* &optional *no-error-p* (*unknown-ok* t)

Returns a host object that recognizes the specified name. If the name is not recognized, it is an error, unless *no-error-p* is non-nil; in that case, nil is returned.

If *unknown-ok* is non-nil (the default), a host table server on the local network is contacted, to see if perhaps it can find the name there. If it can't, an error is signalled or nil is returned, according to *no-error-p*. The host instance created in this manner contains all the kinds of information that a host defined from the host table file has.

If a string of the form CHAOS|*nnn* is used, a host object is created and given *nnn* (interpreted as octal) as its Chaosnet address. This can be done regardless of the *unknown-ok* argument.

The first argument is allowed to be a host object instead of a string. In this case, that argument is simply returned.

sys:unknown-host-name *Condition*

(sys:local-network-error sys:network-error error)

This condition is signaled by si:parse-host when the host is not recognized, if that is an error.

The :name operation on the condition instance returns the string given to si:parse-host.

si:get-host-from-address *address network*

Returns a host object given an address and the name of the network which that address is for. Usually the symbol :chaos is used as the network name.

nil is returned if there is no known host with that address.

fs:get-pathname-host *name* &optional *no-error-p*

Returns a host object that can be used in pathnames. If the name is not recognized, it is an error, unless *no-error-p* is non-nil; in that case, nil is returned.

The first argument is allowed to be a host object instead of a string. In this case, that argument is simply returned.

si:parse-host and fs:get-pathname-host differ in the set of hosts searched.

fs:unknown-pathname-host (fs:pathname-error error) *Condition*

This condition is signaled by fs:get-pathname-host when the host is not recognized, if that is an error.

The :name operation on the condition instance returns the string given to fs:get-pathname-host.

fs:*pathname-host-list* *Variable*

This is a list of all the host objects that support file access.

si:host-alist *Variable*

This variable is a list of one element for each known network host. The element looks like this:

(*full-name host-object* (*nickname nickname2 ... full-name*)
system-type machine-type site
network list-of-addresses network2 list-of-addresses2 ...)

The *full-name* is the host's official name. The :name operation on the host object returns this.

The *host-object* is a flavor instance that represents this host. It may be nil if none has been created yet; si:parse-host creates them when they are referred to.

The *nicknames* are alternate names that si:parse-host should recognize for this host, but which are not its official name.

The *system-type* is a symbol that tells what software the host runs. This is used to decide what flavor of host object to construct. Symbols now used include :lispm, :its, :tops-20, :tenex, :vms, :unix, :multics, :minits, :waits, :chaos-gateway, :dos, :rsx, :magicsix, :msdos, and others. Not all of these are specifically understood in any way by the Lisp Machine. If none of these applies to a host you wish to add, use a new symbol.

The *machine-type* is a symbol that describes the hardware of the host. Symbols in use include :lispm, :pdp10, :pdp11, :vax, :nu, :pe3230, and :ibmpc. (nil) has also been observed to appear here. Note that these machine types attempt to have wide meanings, lumping together various brands, models, etc.

The *site* does not describe anything about the host. Instead it serves to say what the Lisp Machine's site name was when the host was defined. This is so that, when a Lisp Machine system is moved to a different institution that has a disjoint set of hosts, all the old site's hosts can be deleted from the host alist by site reinitialization.

The *network*s and lists of addresses describe how to reach the host. Usually there is only one network and only one address in the list. The generality is so that hosts with multiple addresses on multiple networks can be recorded. Networks include :chaos and :arpa. The address is meaningful only to code for a specific network.

24.8.2 Host Object Operations

:name *Operation on host objects*

Returns the full, official name of the host.

:name-as-file-computer *Operation on host objects*

Returns the name to print in pathnames on this host (assuming it supports files). This is likely to be a short nickname of the host.

:short-name *Operation on host objects*

Returns the shortest known nickname for this host.

:pathname-host-namep *string* *Operation on host objects*

Returns t if *string* is recognized as a name for this host for purposes of pathname parsing. The local host will recognise LM as a pathname host name.

:system-type *Operation on host objects*

Returns the operating system type symbol for this host. See page 810.

:network-type *Operation on host objects*

Returns the symbol for one network that this host is connected to, or nil if it is not connected to any. :chaos is preferred if it is one of the possible values.

:network-typep *network* *Operation on host objects*

Returns t if the host is connected to the specified network.

:network-addresses *Operation on host objects*

Returns an alternating list of network names and lists of addresses, such as

```
(:chaos (3104) :arpa (106357002))
```

You can therefore find out all networks a host is known to be on, and its addresses on any network.

:sample-pathname *Operation on host objects*

Returns a pathname for this host, whose device, directory, name, type and version components are all nil. Sample pathnames are often useful because many file-system-dependent pathname operations depend only on the pathname's host.

:open-streams *Operation on host objects*

Returns a list of all the open file streams for files on this host.

:close-all-files *Operation on host objects*

Closes all file streams open for files on this host.

:generic-base-type *type-component* *Operation on host objects*

Returns the type component for a generic pathname assuming it is being made from a pathname whose type component is the one specified.