Programming Language—Common Lisp

24. System Construction

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

24.1 System Construction Concepts

24.1.1 Loading

To **load** a *file* is to treat its contents as *code* and *execute* that *code*. The *file* may contain ***source code*** or ***compiled code***.

A *file* containing *source code* is called a ***source file***. *Loading* a *source file* is accomplished essentially by sequentially *reading*$_2$ the *forms* in the file, *evaluating* each immediately after it is *read*.

A *file* containing *compiled code* is called a ***compiled file***. *Loading* a *compiled file* is similar to *loading* a *source file*, except that the *file* does not contain text but rather an *implementation-dependent* representation of pre-digested *expressions* created by the *compiler*. Often, a *compiled file* can be *loaded* more quickly than a *source file*. See Section 3.2 (Compilation).

The way in which a *source file* is distinguished from a *compiled file* is *implementation-dependent*.

24.1.2 Features

A ***feature*** is an aspect or attribute of Common Lisp, of the *implementation*, or of the *environment*. A *feature* is identified by a *symbol*.

A *feature* is said to be ***present*** in a *Lisp image* if and only if the *symbol* naming it is an *element* of the *list* held by the *variable* ***features***, which is called the ***features list***.

24.1.2.1 Feature Expressions

Boolean combinations of *features*, called ***feature expressions***, are used by the `#+` and `#-` *reader macros* in order to direct conditional *reading* of *expressions* by the *Lisp reader*.

The rules for interpreting a *feature expression* are as follows:

feature

If a *symbol* naming a *feature* is used as a *feature expression*, the *feature expression* succeeds if that *feature* is *present*; otherwise it fails.

(not *feature-conditional*)

A **not** *feature expression* succeeds if its argument *feature-conditional* fails; otherwise, it succeeds.

(and {*feature-conditional*}*)

An **and** *feature expression* succeeds if all of its argument *feature-conditionals* succeed; otherwise, it fails.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

(or {*feature-conditional*}*)

An **or** *feature expression* succeeds if any of its argument *feature-conditionals* succeed; otherwise, it fails.

24.1.2.1.1 Examples of Feature Expressions

For example, suppose that in *implementation* A, the *features* `spice` and `perq` are *present*, but the *feature* `lispm` is not *present*; in *implementation* B, the feature `lispm` is *present*, but the *features* `spice` and `perq` are not *present*; and in *implementation* C, none of the features `spice`, *lispm*, or `perq` are *present*. Figure 24–1 shows some sample *expressions*, and how they would be *read*$_2$ in these *implementations*.

`(cons #+spice "Spice" #-spice "Lispm" x)`	
in *implementation* A ...	`(CONS "Spice" X)`
in *implementation* B ...	`(CONS "Lispm" X)`
in *implementation* C ...	`(CONS "Lispm" X)`
`(cons #+spice "Spice" #+LispM "Lispm" x)`	
in *implementation* A ...	`(CONS "Spice" X)`
in *implementation* B ...	`(CONS "Lispm" X)`
in *implementation* C ...	`(CONS X)`
`(setq a '(1 2 #+perq 43 #+(not perq) 27))`	
in *implementation* A ...	`(SETQ A '(1 2 43))`
in *implementation* B ...	`(SETQ A '(1 2 27))`
in *implementation* C ...	`(SETQ A '(1 2 27))`
`(let ((a 3) #+(or spice lispm) (b 3)) (foo a))`	
in *implementation* A ...	`(LET ((A 3) (B 3)) (FOO A))`
in *implementation* B ...	`(LET ((A 3) (B 3)) (FOO A))`
in *implementation* C ...	`(LET ((A 3)) (FOO A))`
`(cons #+Lispm "#+Spice" #+Spice "foo" #-(or Lispm Spice) 7 x)`	
in *implementation* A ...	`(CONS "foo" X)`
in *implementation* B ...	`(CONS "#+Spice" X)`
in *implementation* C ...	`(CONS 7 X)`

Figure 24–1. Features examples

compile-file *Function*

Syntax:

compile-file *input-file* &key *output-file verbose print external-format*

→ *output-truename, warnings-p, failure-p*

Arguments and Values:

input-file—a *pathname designator*. (Default fillers for unspecified components are taken from ***default-pathname-defaults***.)

output-file—a *pathname designator*. The default is *implementation-defined*.

verbose—a *generalized boolean*. The default is the *value* of ***compile-verbose***.

print—a *generalized boolean*. The default is the *value* of ***compile-print***.

external-format—an *external file format designator*. The default is `:default`.

output-truename—a *pathname* (the **truename** of the output *file*), or **nil**.

warnings-p—a *generalized boolean*.

failure-p—a *generalized boolean*.

Description:

compile-file transforms the contents of the file specified by *input-file* into *implementation-dependent* binary data which are placed in the file specified by *output-file*.

The *file* to which *input-file* refers should be a *source file*. *output-file* can be used to specify an output *pathname*; the actual *pathname* of the *compiled file* to which *compiled code* will be output is computed as if by calling **compile-file-pathname**.

If *input-file* or *output-file* is a *logical pathname*, it is translated into a *physical pathname* as if by calling **translate-logical-pathname**.

If *verbose* is *true*, **compile-file** prints a message in the form of a comment (*i.e.*, with a leading *semicolon*) to *standard output* indicating what *file* is being *compiled* and other useful information. If *verbose* is *false*, **compile-file** does not print this information.

If *print* is *true*, information about *top level forms* in the file being compiled is printed to *standard output*. Exactly what is printed is *implementation-dependent*, but nevertheless some information is printed. If *print* is **nil**, no information is printed.

The *external-format* specifies the *external file format* to be used when opening the *file*; see the

function **open**. **compile-file** and **load** must cooperate in such a way that the resulting *compiled file* can be *loaded* without specifying an *external file format* anew; see the *function* **load**.

compile-file binds ***readtable*** and ***package*** to the values they held before processing the file.

compile-file-truename is bound by **compile-file** to hold the *truename* of the *pathname* of the file being compiled.

compile-file-pathname is bound by **compile-file** to hold a *pathname* denoted by the first argument to **compile-file**, merged against the defaults; that is, `(pathname (merge-pathnames` *input-file*`))`.

The compiled *functions* contained in the *compiled file* become available for use when the *compiled file* is *loaded* into Lisp. Any function definition that is processed by the compiler, including `#'(lambda ...)` forms and local function definitions made by **flet**, **labels** and **defun** forms, result in an *object* of *type* **compiled-function**.

The *primary value* returned by **compile-file**, *output-truename*, is the **truename** of the output file, or **nil** if the file could not be created.

The *secondary value*, *warnings-p*, is *false* if no *conditions* of *type* **error** or **warning** were detected by the compiler, and *true* otherwise.

The *tertiary value*, *failure-p*, is *false* if no *conditions* of *type* **error** or **warning** (other than **style-warning**) were detected by the compiler, and *true* otherwise.

For general information about how *files* are processed by the *file compiler*, see Section 3.2.3 (File Compilation).

Programs to be compiled by the *file compiler* must only contain *externalizable objects*; for details on such *objects*, see Section 3.2.4 (Literal Objects in Compiled Files). For information on how to extend the set of *externalizable objects*, see the *function* **make-load-form** and Section 3.2.4.4 (Additional Constraints on Externalizable Objects).

Affected By:

error-output, ***standard-output***, ***compile-verbose***, ***compile-print***

The computer's file system.

Exceptional Situations:

For information about errors detected during the compilation process, see Section 3.2.5 (Exceptional Situations in the Compiler).

An error of *type* **file-error** might be signaled if `(wild-pathname-p` *input-file*`)` returns true.

If either the attempt to open the *source file* for input or the attempt to open the *compiled file* for output fails, an error of *type* **file-error** is signaled.

See Also:

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

compile-file-pathname *Function*

Syntax:

compile-file-pathname *input-file* `&key` *output-file* `&allow-other-keys` → *pathname*

Arguments and Values:

input-file—a *pathname designator*. (Default fillers for unspecified components are taken from ***default-pathname-defaults***.)

output-file—a *pathname designator*. The default is *implementation-defined*.

pathname—a *pathname*.

Description:

Returns the *pathname* that **compile-file** would write into, if given the same arguments.

The defaults for the *output-file* are taken from the *pathname* that results from merging the *input-file* with the *value* of ***default-pathname-defaults***, except that the type component should default to the appropriate *implementation-defined* default type for *compiled files*.

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

If an implementation supports additional keyword arguments to **compile-file**, **compile-file-pathname** must accept the same arguments.

Examples:

See **logical-pathname-translations**.

Exceptional Situations:

An error of *type* **file-error** might be signaled if either *input-file* or *output-file* is *wild*.

See Also:

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

load *Function*

Syntax:

load *filespec* `&key` *verbose print*
if-does-not-exist external-format

→ *generalized-boolean*

Arguments and Values:

filespec—a *stream*, or a *pathname designator*. The default is taken from ***default-pathname-defaults***.

verbose—a *generalized boolean*. The default is the *value* of ***load-verbose***.

print—a *generalized boolean*. The default is the *value* of ***load-print***.

if-does-not-exist—a *generalized boolean*. The default is *true*.

external-format—an *external file format designator*. The default is `:default`.

generalized-boolean—a *generalized boolean*.

Description:

load *loads* the *file* named by *filespec* into the Lisp environment.

The manner in which a *source file* is distinguished from a *compiled file* is *implementation-dependent*. If the file specification is not complete and both a *source file* and a *compiled file* exist which might match, then which of those files **load** selects is *implementation-dependent*.

If *filespec* is a *stream*, **load** determines what kind of *stream* it is and loads directly from the *stream*. If *filespec* is a *logical pathname*, it is translated into a *physical pathname* as if by calling **translate-logical-pathname**.

load sequentially executes each *form* it encounters in the *file* named by *filespec*. If the *file* is a *source file* and the *implementation* chooses to perform *implicit compilation*, **load** must recognize *top level forms* as described in Section 3.2.3.1 (Processing of Top Level Forms) and arrange for each *top level form* to be executed before beginning *implicit compilation* of the next. (Note, however, that processing of **eval-when** *forms* by **load** is controlled by the `:execute` situation.)

If *verbose* is *true*, **load** prints a message in the form of a comment (*i.e.*, with a leading *semicolon*) to *standard output* indicating what *file* is being *loaded* and other useful information. If *verbose* is *false*, **load** does not print this information.

If *print* is *true*, **load** incrementally prints information to *standard output* showing the progress of the *loading* process. For a *source file*, this information might mean printing the *values yielded* by each *form* in the *file* as soon as those *values* are returned. For a *compiled file*, what is printed might not reflect precisely the contents of the *source file*, but some information is generally printed. If *print* is *false*, **load** does not print this information.

If the file named by *filespec* is successfully loaded, **load** returns *true*.

If the file does not exist, the specific action taken depends on *if-does-not-exist*: if it is **nil**, **load** returns **nil**; otherwise, **load** signals an error.

The *external-format* specifies the *external file format* to be used when opening the *file* (see the *function* **open**), except that when the *file* named by *filespec* is a *compiled file*, the *external-format* is ignored. **compile-file** and **load** cooperate in an *implementation-dependent* way to assure the preservation of the *similarity* of *characters* referred to in the *source file* at the time the *source file* was processed by the *file compiler* under a given *external file format*, regardless of the value of *external-format* at the time the *compiled file* is *loaded*.

load binds ***readtable*** and ***package*** to the values they held before *loading* the file.

load-truename is *bound* by **load** to hold the *truename* of the *pathname* of the file being *loaded*.

load-pathname is *bound* by **load** to hold a *pathname* that represents *filespec* merged against the defaults. That is, (pathname (merge-pathnames *filespec*)).

Examples:

```
;Establish a data file...
 (with-open-file (str "data.in" :direction :output :if-exists :error)
   (print 1 str) (print '(setq a 888) str) t)
→ T
 (load "data.in") → true
 a → 888
 (load (setq p (merge-pathnames "data.in")) :verbose t)
; Loading contents of file /fred/data.in
; Finished loading /fred/data.in
→ true
 (load p :print t)
; Loading contents of file /fred/data.in
;  1
;  888
; Finished loading /fred/data.in
→ true

 ;----[Begin file SETUP]----
 (in-package "MY-STUFF")
 (defmacro compile-truename () `',*compile-file-truename*)
 (defvar *my-compile-truename* (compile-truename) "Just for debugging.")
 (defvar *my-load-pathname* *load-pathname*)
 (defun load-my-system ()
   (dolist (module-name '("FOO" "BAR" "BAZ"))
     (load (merge-pathnames module-name *my-load-pathname*))))
 ;----[End of file SETUP]----

 (load "SETUP")
 (load-my-system)
```


Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

Affected By:

The implementation, and the host computer's file system.

Exceptional Situations:

If :if-does-not-exist is supplied and is *true*, or is not supplied, **load** signals an error of *type* **file-error** if the file named by *filespec* does not exist, or if the *file system* cannot perform the requested operation.

An error of *type* **file-error** might be signaled if (wild-pathname-p *filespec*) returns *true*.

See Also:

error, **merge-pathnames**, ***load-verbose***, ***default-pathname-defaults***, **pathname**, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

with-compilation-unit *Macro*

Syntax:

with-compilation-unit (⟦↓*option*⟧) {*form*}* → {*result*}*

option::=:override *override*

Arguments and Values:

override—a *generalized boolean*; evaluated. The default is **nil**.

forms—an *implicit progn*.

results—the *values* returned by the *forms*.

Description:

Executes *forms* from left to right. Within the *dynamic environment* of **with-compilation-unit**, actions deferred by the compiler until the end of compilation will be deferred until the end of the outermost call to **with-compilation-unit**.

The set of *options* permitted may be extended by the implementation, but the only *standardized* keyword is `:override`.

If nested dynamically only the outer call to **with-compilation-unit** has any effect unless the value associated with `:override` is *true*, in which case warnings are deferred only to the end of the innermost call for which *override* is *true*.

The function **compile-file** provides the effect of

```
(with-compilation-unit (:override nil) ...)
```

around its *code*.

Any *implementation-dependent* extensions can only be provided as the result of an explicit programmer request by use of an *implementation-dependent* keyword. *Implementations* are forbidden from attaching additional meaning to a use of this macro which involves either no keywords or just the keyword `:override`.

Examples:

If an *implementation* would normally defer certain kinds of warnings, such as warnings about undefined functions, to the end of a compilation unit (such as a *file*), the following example shows how to cause those warnings to be deferred to the end of the compilation of several files.

```
(defun compile-files (&rest files)
  (with-compilation-unit ()
    (mapcar #'(lambda (file) (compile-file file)) files)))

(compile-files "A" "B" "C")
```

Note however that if the implementation does not normally defer any warnings, use of *with-compilation-unit* might not have any effect.

See Also:

compile, **compile-file**

∗features∗

∗features∗ *Variable*

Value Type:

a *proper list*.

Initial Value:

implementation-dependent.

Description:

The *value* of ***features*** is called the *features list*. It is a *list* of *symbols*, called *features*, that correspond to some aspect of the *implementation* or *environment*.

Most *features* have *implementation-dependent* meanings; The following meanings have been assigned to feature names:

`:cltl1`

If present, indicates that the `LISP` *package purports to conform* to the 1984 specification *Common Lisp: The Language*. It is possible, but not required, for a *conforming implementation* to have this feature because this specification specifies that its *symbols* are to be in the `COMMON-LISP` *package*, not the `LISP` package.

`:cltl2`

If present, indicates that the implementation *purports to conform* to *Common Lisp: The Language, Second Edition*. This feature must not be present in any *conforming implementation*, since conformance to that document is not compatible with conformance to this specification. The name, however, is reserved by this specification in order to help programs distinguish implementations which conform to that document from implementations which conform to this specification.

`:ieee-floating-point`

If present, indicates that the implementation *purports to conform* to the requirements of *IEEE Standard for Binary Floating-Point Arithmetic*.

`:x3j13`

If present, indicates that the implementation conforms to some particular working draft of this specification, or to some subset of features that approximates a belief about what this specification might turn out to contain. A *conforming implementation* might or might not contain such a feature. (This feature is intended primarily as a stopgap in order to provide implementors something to use prior to the availability of a draft standard, in order to discourage them from introducing the `:draft-ansi-cl` and `:ansi-cl` *features* prematurely.)

`:draft-ansi-cl`

If present, indicates that the *implementation purports to conform* to the first full draft of this specification, which went to public review in 1992. A *conforming implementation* which has the `:draft-ansi-cl-2` or `:ansi-cl` *feature* is not permitted to retain the `:draft-ansi-cl` *feature* since incompatible changes were made subsequent to the first draft.

`:draft-ansi-cl-2`

If present, indicates that a second full draft of this specification has gone to public review, and that the *implementation purports to conform* to that specification. (If additional public review drafts are produced, this keyword will continue to refer to the second draft, and additional keywords will be added to identify conformance with such later drafts. As such, the meaning of this keyword can be relied upon not to change over time.) A *conforming implementation* which has the `:ansi-cl` *feature* is only permitted to retain the `:draft-ansi-cl` *feature* if the finally approved standard is not incompatible with the draft standard.

`:ansi-cl`

If present, indicates that this specification has been adopted by ANSI as an official standard, and that the *implementation purports to conform*.

`:common-lisp`

This feature must appear in ***features*** for any implementation that has one or more of the features `:x3j13`, `:draft-ansi-cl`, or `:ansi-cl`. It is intended that it should also appear in implementations which have the features `:cltl1` or `:cltl2`, but this specification cannot force such behavior. The intent is that this feature should identify the language family named "Common Lisp," rather than some specific dialect within that family.

See Also:

Section 1.5.2.1.1 (Use of Read-Time Conditionals), Section 2.4 (Standard Macro Characters)

Notes:

The *value* of ***features*** is used by the `#+` and `#-` reader syntax.

Symbols in the *features list* may be in any *package*, but in practice they are generally in the `KEYWORD` *package*. This is because `KEYWORD` is the *package* used by default when *reading*$_2$ *feature expressions* in the `#+` and `#-` *reader macros*. *Code* that needs to name a *feature*$_2$ in a *package* P (other than `KEYWORD`) can do so by making explicit use of a *package prefix* for P, but note that such *code* must also assure that the *package* P exists in order for the *feature expression* to be *read*$_2$—even in cases where the *feature expression* is expected to fail.

It is generally considered wise for an *implementation* to include one or more *features* identifying

the specific *implementation*, so that conditional expressions can be written which distinguish idiosyncrasies of one *implementation* from those of another. Since features are normally *symbols* in the `KEYWORD` *package* where name collisions might easily result, and since no uniquely defined mechanism is designated for deciding who has the right to use which *symbol* for what reason, a conservative strategy is to prefer names derived from one's own company or product name, since those names are often trademarked and are hence less likely to be used unwittingly by another *implementation*.

compile-file-pathname, *compile-file-truename*

Variable

Value Type:

The *value* of ***compile-file-pathname*** must always be a *pathname* or **nil**. The *value* of ***compile-file-truename*** must always be a *physical pathname* or **nil**.

Initial Value:

nil.

Description:

During a call to **compile-file**, ***compile-file-pathname*** is *bound* to the *pathname* denoted by the first argument to **compile-file**, merged against the defaults; that is, it is *bound* to `(pathname (merge-pathnames` *input-file*`))`. During the same time interval, ***compile-file-truename*** is *bound* to the *truename* of the *file* being *compiled*.

At other times, the *value* of these *variables* is **nil**.

If a *break loop* is entered while **compile-file** is ongoing, it is *implementation-dependent* whether these *variables* retain the *values* they had just prior to entering the *break loop* or whether they are *bound* to **nil**.

The consequences are unspecified if an attempt is made to *assign* or *bind* either of these *variables*.

Affected By:

The *file system*.

See Also:

compile-file

∗load-pathname∗, ∗load-truename∗ *Variable*

Value Type:

The *value* of ***load-pathname*** must always be a *pathname* or **nil**. The *value* of ***load-truename*** must always be a *physical pathname* or **nil**.

Initial Value:

nil.

Description:

During a call to **load**, ***load-pathname*** is *bound* to the *pathname* denoted by the the first argument to **load**, merged against the defaults; that is, it is *bound* to `(pathname (merge-pathnames filespec))`. During the same time interval, ***load-truename*** is *bound* to the *truename* of the *file* being loaded.

At other times, the *value* of these *variables* is **nil**.

If a *break loop* is entered while **load** is ongoing, it is *implementation-dependent* whether these *variables* retain the *values* they had just prior to entering the *break loop* or whether they are *bound* to **nil**.

The consequences are unspecified if an attempt is made to *assign* or *bind* either of these *variables*.

Affected By:

The *file system*.

See Also:

load

∗compile-print∗, ∗compile-verbose∗ *Variable*

Value Type:

a *generalized boolean*.

Initial Value:

implementation-dependent.

Description:

The *value* of ***compile-print*** is the default value of the `:print` *argument* to **compile-file**. The *value* of ***compile-verbose*** is the default value of the `:verbose` *argument* to **compile-file**.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

See Also:

compile-file

∗load-print∗, ∗load-verbose∗ *Variable*

Value Type:

a *generalized boolean*.

Initial Value:

The initial *value* of ***load-print*** is *false*. The initial *value* of ***load-verbose*** is *implementation-dependent*.

Description:

The *value* of ***load-print*** is the default value of the `:print` *argument* to **load**. The *value* of ***load-verbose*** is the default value of the `:verbose` *argument* to **load**.

See Also:

load

∗modules∗ *Variable*

Value Type:

a *list* of *strings*.

Initial Value:

implementation-dependent.

Description:

The *value* of ***modules*** is a list of names of the modules that have been loaded into the current *Lisp image*.

Affected By:

provide

See Also:

provide, **require**

Notes:

The variable ***modules*** is deprecated.

provide, require *Function*

Syntax:

provide *module-name* → *implementation-dependent*

require *module-name* &optional *pathname-list* → *implementation-dependent*

Arguments and Values:

module-name—a *string designator*.

pathname-list—**nil**, or a *designator* for a *non-empty list* of *pathname designators*. The default is **nil**.

Description:

provide adds the *module-name* to the *list* held by ***modules***, if such a name is not already present.

require tests for the presence of the *module-name* in the *list* held by ***modules***. If it is present, **require** immediately returns. Otherwise, an attempt is made to load an appropriate set of *files* as follows: The *pathname-list* argument, if *non-nil*, specifies a list of *pathnames* to be loaded in order, from left to right. If the *pathname-list* is **nil**, an *implementation-dependent* mechanism will be invoked in an attempt to load the module named *module-name*; if no such module can be loaded, an error of *type* **error** is signaled.

Both functions use **string=** to test for the presence of a *module-name*.

Examples:

```
;;; This illustrates a nonportable use of REQUIRE, because it
;;; depends on the implementation-dependent file-loading mechanism.

(require "CALCULUS")

;;; This use of REQUIRE is nonportable because of the literal
;;; physical pathname.

(require "CALCULUS" "/usr/lib/lisp/calculus")

;;; One form of portable usage involves supplying a logical pathname,
;;; with appropriate translations defined elsewhere.

(require "CALCULUS" "lib:calculus")
```



```
;;; Another form of portable usage involves using a variable or
;;; table lookup function to determine the pathname, which again
;;; must be initialized elsewhere.

(require "CALCULUS" *calculus-module-pathname*)
```

Side Effects:

provide modifies ***modules***.

Affected By:

The specific action taken by **require** is affected by calls to **provide** (or, in general, any changes to the *value* of ***modules***).

Exceptional Situations:

Should signal an error of *type* **type-error** if *module-name* is not a *string designator*.

If **require** fails to perform the requested operation due to a problem while interacting with the *file system*, an error of *type* **file-error** is signaled.

An error of *type* **file-error** might be signaled if any *pathname* in *pathname-list* is a *designator* for a *wild pathname*.

See Also:

modules, Section 19.1.2 (Pathnames as Filenames)

Notes:

The functions **provide** and **require** are deprecated.

If a module consists of a single *package*, it is customary for the package and module names to be the same.