15. Practical: A Portable Pathname Library

As I discussed in the previous chapter, Common Lisp provides an abstraction, the pathname, that's supposed to insulate you from the details of how different operating systems and file systems name files. Pathnames provide a useful API for manipulating names as names, but when it comes to the functions that actually interact with the file system, things get a bit hairy.

The root of the problem, as I mentioned, is that the pathname abstraction was designed to represent filenames on a much wider variety of file systems than are commonly used now. Unfortunately, by making pathnames abstract enough to account for a wide variety of file systems, Common Lisp's designers left implementers with a fair number of choices to make about how exactly to map the pathname abstraction onto any particular file system. Consequently, different implementers, each implementing the pathname abstraction for the same file system, just by making different choices at a few key junctions, could end up with conforming implementations that nonetheless provide different behavior for several of the main pathname-related functions.

However, one way or another, all implementations provide the same basic functionality, so it's not too hard to write a library that provides a consistent interface for the most common operations across different implementations. That's your task for this chapter. In addition to giving you several useful functions that you'll use in future chapters, writing this library will give you a chance to learn how to write code that deals with differences between implementations.

The API

The basic operations the library will support will be getting a list of files in a directory and determining whether a file or directory with a given name exists. You'll also write a function for recursively walking a directory hierarchy, calling a given function for each pathname in the tree.

In theory, these directory listing and file existence operations are already provided by the standard functions **`DIRECTORY`** and **`PROBE-FILE`**. However, as you'll see, there are enough different ways to implement these functions--all within the bounds of valid interpretations of the language standard--that you'll want to write new functions that provide a consistent behavior across implementations.

FEATURES and Read-Time Conditionalization

Before you can implement this API in a library that will run correctly on multiple Common Lisp implementations, I need to show you the mechanism for writing implementation-specific code.

While most of the code you write can be "portable" in the sense that it will run the same on any conforming Common Lisp implementation, you may occasionally need to rely on implementation-specific functionality or to write slightly different bits of code for different implementations. To allow you to do so without totally destroying the portability of your code, Common Lisp provides a mechanism, called *read-time conditionalization*, that allows you to conditionally include code based on various features such as what implementation it's being run in.

The mechanism consists of a variable **`*FEATURES*`** and two extra bits of syntax understood by the Lisp reader. **`*FEATURES*`** is a list of symbols; each symbol represents a "feature" that's present in the implementation or on the underlying platform. These symbols are then used in *feature expressions* that evaluate to true or false depending on whether the symbols in the expression are present in **`*FEATURES*`**. The simplest feature expression is a single symbol; the expression is true if the symbol is in **`*FEATURES*`** and false if it isn't. Other feature expressions are boolean expressions built out of **`NOT`**, **`AND`**, and **`OR`** operators. For instance, if you wanted to conditionalize some code to be included only if the features `foo` and `bar` were present, you could write the feature expression `(and foo bar).`

The reader uses feature expressions in conjunction with two bits of syntax, `#+` and `#-`. When the reader sees either of these bits of syntax, it first reads a feature expression and then evaluates it as I just described. When a feature expression following a `#+` is true, the reader reads the next expression normally. Otherwise it skips the next expression, treating it as whitespace. `#-` works the same way except it reads the form if the feature expression is false and skips it if it's true.

The initial value of **`*FEATURES*`** is implementation dependent, and what functionality is implied by the presence of any given symbol is likewise defined by the implementation. However, all implementations include at least one symbol that indicates what implementation it is. For instance, Allegro Common Lisp includes the symbol `:allegro`, CLISP includes `:clisp`, SBCL includes `:sbcl`, and CMUCL includes `:cmu`. To avoid dependencies on packages that may or may not exist in different implementations, the symbols in **`*FEATURES*`** are usually keywords, and the reader binds **`*PACKAGE*`** to the **`KEYWORD`** package while reading feature expressions. Thus, a name with no package qualification will be read as a keyword symbol. So, you could write a function that behaves slightly differently in each of the implementations just mentioned like this:

```
(defun foo ()
  #+allegro (do-one-thing)
  #+sbcl (do-another-thing)
  #+clisp (something-else)
```

```
      #+cmu (yet-another-version)
      #-(or allegro sbcl clisp cmu) (error "Not implemented"))
```

In Allegro that code will be read as if it had been written like this:

```
(defun foo ()
  (do-one-thing))
```

while in SBCL the reader will read this:

```
(defun foo ()
  (do-another-thing))
```

while in an implementation other than one of the ones specifically conditionalized, it will read this:

```
(defun foo ()
  (error "Not implemented"))
```

Because the conditionalization happens in the reader, the compiler doesn't even see expressions that are skipped.[1] This means you pay no runtime cost for having different versions for different implementations. Also, when the reader skips conditionalized expressions, it doesn't bother interning symbols, so the skipped expressions can safely contain symbols from packages that may not exist in other implementations.

Packaging the Library

Speaking of packages, if you download the complete code for this library, you'll see that it's defined in a new package, `com.gigamonkeys.pathnames`. I'll discuss the details of defining and using packages in Chapter 21. For now you should note that some implementations provide their own packages that contain functions with some of the same names as the ones you'll define in this chapter and make those names available in the CL-USER package. Thus, if you try to define the functions from this library while in the CL-USER package, you may get errors or warnings about clobbering existing definitions. To avoid this possibility, you can create a file called `packages.lisp` with the following contents:

```
(in-package :cl-user)

(defpackage :com.gigamonkeys.pathnames
  (:use :common-lisp)
  (:export
   :list-directory
   :file-exists-p
   :directory-pathname-p
   :file-pathname-p
   :pathname-as-directory
   :pathname-as-file
   :walk-directory
   :directory-p
   :file-p))
```

and **`LOAD`** it. Then at the REPL or at the top of the file where you type the definitions from this chapter, type the following expression:

```
(in-package :com.gigamonkeys.pathnames)
```

In addition to avoiding name conflicts with symbols already available in `CL-USER`, packaging the library this way also makes it easier to use in other code, as you'll see in several future chapters.

Listing a Directory

You can implement the function for listing a single directory, `list-directory`, as a thin wrapper around the standard function **`DIRECTORY`**. **`DIRECTORY`** takes a special kind of pathname, called a *wild pathname*, that has one or more components containing the special value `:wild` and returns a list of pathnames representing files in the file system that match the wild pathname.[2] The matching algorithm--like most things having to do with the interaction between Lisp and a particular file system--isn't defined by the language standard, but most implementations on Unix and Windows follow the same basic scheme.

The **`DIRECTORY`** function has two problems that you need to address with `list-directory`. The main one is that certain aspects of its behavior differ fairly significantly between different Common Lisp implementations, even on the same operating system. The other is that while **`DIRECTORY`** provides a powerful interface for listing files, to use it properly requires understanding some rather subtle points about the pathname abstraction. Between these subtleties and the idiosyncrasies of different implementations, actually writing portable code that uses **`DIRECTORY`** to do something as simple as listing all the files and subdirectories in a single directory can be a frustrating experience. You can deal with those subtleties and idiosyncrasies once and for all, by writing `list-directory`, and forget them thereafter.

One subtlety I discussed in Chapter 14 is the two ways to represent the name of a directory as a pathname: directory form and file form.

To get **`DIRECTORY`** to return a list of files in `/home/peter/`, you need to pass it a wild pathname whose directory component is the directory you want to list and whose name and type components are `:wild`. Thus, to get a listing of the files in `/home/peter/`, it might seem you could write this:

```
(directory (make-pathname :name :wild :type :wild :defaults home-dir))
```

where `home-dir` is a pathname representing `/home/peter/`. This would work if `home-dir` were in directory form. But if it were in file form--for example, if it had been created by parsing the namestring `"/home/peter"`--then that same expression would list all the files in `/home` since the name component `"peter"` would be replaced with `:wild`.

To avoid having to worry about explicitly converting between representations, you can define `list-directory` to accept a nonwild pathname in either form, which it will then convert to the appropriate wild pathname.

To help with this, you should define a few helper functions. One, `component-present-p`, will test whether a given component of a pathname is "present," meaning neither **`NIL`** nor the special value `:unspecific`.[3] Another, `directory-pathname-p`, tests whether a pathname is already in directory form, and the third, `pathname-as-directory`, converts any pathname to a directory form pathname.

```
(defun component-present-p (value)
  (and value (not (eql value :unspecific))))

(defun directory-pathname-p  (p)
  (and
   (not (component-present-p (pathname-name p)))
   (not (component-present-p (pathname-type p)))
   p))

(defun pathname-as-directory (name)
  (let ((pathname (pathname name)))
    (when (wild-pathname-p pathname)
      (error "Can't reliably convert wild pathnames."))
    (if (not (directory-pathname-p name))
      (make-pathname
       :directory (append (or (pathname-directory pathname) (list :relative))
                          (list (file-namestring pathname)))
       :name      nil
       :type      nil
       :defaults pathname)
      pathname)))
```

Now it seems you could generate a wild pathname to pass to **`DIRECTORY`** by calling **`MAKE-PATHNAME`** with a directory form name returned by `pathname-as-directory`. Unfortunately, it's not quite that simple, thanks to a quirk in CLISP's implementation of **`DIRECTORY`**. In CLISP, **`DIRECTORY`** won't return files with no extension unless the type component of the wildcard is **`NIL`** rather than `:wild`. So you can define a function, `directory-wildcard`, that takes a pathname in either directory or file form and returns a proper wildcard for the given implementation using read-time conditionalization to make a pathname with a `:wild` type component in all implementations except for CLISP and **`NIL`** in CLISP.

```
(defun directory-wildcard (dirname)
  (make-pathname
   :name :wild
   :type #-clisp :wild #+clisp nil
   :defaults (pathname-as-directory dirname)))
```

Note how each read-time conditional operates at the level of a single expression After `#-clisp`, the expression `:wild` is either read or skipped; likewise, after `#+clisp`, the **`NIL`** is read or skipped.

Now you can take a first crack at the `list-directory` function.

```
(defun list-directory (dirname)
  (when (wild-pathname-p dirname)
    (error "Can only list concrete directory names."))
  (directory (directory-wildcard dirname)))
```

As it stands, this function would work in SBCL, CMUCL, and LispWorks. Unfortunately, a couple more implementation differences remain to be smoothed over. One is that not all implementations will return subdirectories of the given directory. Allegro, SBCL, CMUCL, and

LispWorks do. OpenMCL doesn't by default but will if you pass **DIRECTORY** a true value via the implementation-specific keyword argument `:directories`. CLISP's **DIRECTORY** returns subdirectories only when it's passed a wildcard pathname with `:wild` as the last element of the directory component and **NIL** name and type components. In this case, it returns *only* subdirectories, so you'll need to call **DIRECTORY** twice with different wildcards and combine the results.

Once you get all the implementations returning directories, you'll discover they can also differ in whether they return the names of directories in directory or file form. You want `list-directory` to always return directory names in directory form so you can differentiate subdirectories from regular files based on just the name. Except for Allegro, all the implementations this library will support do that. Allegro, on the other hand, requires you to pass **DIRECTORY** the implementation-specific keyword argument `:directories-are-files` **NIL** to get it to return directories in file form.

Once you know how to make each implementation do what you want, actually writing `list-directory` is simply a matter of combining the different versions using read-time conditionals.

```
(defun list-directory (dirname)
  (when (wild-pathname-p dirname)
    (error "Can only list concrete directory names."))
  (let ((wildcard (directory-wildcard dirname)))

    #+(or sbcl cmu lispworks)
    (directory wildcard)

    #+openmcl
    (directory wildcard :directories t)

    #+allegro
    (directory wildcard :directories-are-files nil)

    #+clisp
    (nconc
     (directory wildcard)
     (directory (clisp-subdirectories-wildcard wildcard)))

    #-(or sbcl cmu lispworks openmcl allegro clisp)
    (error "list-directory not implemented")))
```

The function `clisp-subdirectories-wildcard` isn't actually specific to CLISP, but since it isn't needed by any other implementation, you can guard its definition with a read-time conditional. In this case, since the expression following the #+ is the whole **DEFUN**, the whole function definition will be included or not, depending on whether `clisp` is present in ***FEATURES***.

```
#+clisp
(defun clisp-subdirectories-wildcard (wildcard)
  (make-pathname
   :directory (append (pathname-directory wildcard) (list :wild))
   :name nil
   :type nil
   :defaults wildcard))
```

Testing a File's Existence

To replace **PROBE-FILE**, you can define a function called `file-exists-p`. It should accept a pathname and return an equivalent pathname if the file exists and **NIL** if it doesn't. It should be able to accept the name of a directory in either directory or file form but should always return a directory form pathname if the file exists and is a directory. This will allow you to use `file-exists-p`, along with `directory-pathname-p`, to test whether an arbitrary name is the name of a file or directory.

In theory, `file-exists-p` is quite similar to the standard function **PROBE-FILE**; indeed, in several implementations--SBCL, LispWorks, and OpenMCL--**PROBE-FILE** already gives you the behavior you want for `file-exists-p`. But not all implementations of **PROBE-FILE** behave quite the same.

Allegro and CMUCL's **PROBE-FILE** functions are close to what you need--they will accept the name of a directory in either form but, instead of returning a directory form name, simply return the name in the same form as the argument it was passed. Luckily, if passed the name of a nondirectory in directory form, they return **NIL**. So with those implementations you can get the behavior you want by first passing the name to **PROBE-FILE** in directory form--if the file exists and is a directory, it will return the directory form name. If that call returns **NIL**, then you try again with a file form name.

CLISP, on the other hand, once again has its own way of doing things. Its **PROBE-FILE** immediately signals an error if passed a name in directory form, regardless of whether a file or directory exists with that name. It also signals an error if passed a name in file form that's actually the name of a directory. For testing whether a directory exists, CLISP provides its own function: `probe-directory` (in the `ext` package). This is almost the mirror image of **PROBE-FILE**: it signals an error if passed a name in file form or if passed a name in directory form that happens to name a file. The only difference is it returns **T** rather than a pathname when the named directory exists.

But even in CLISP you can implement the desired semantics by wrapping the calls to **PROBE-FILE** and `probe-directory` in **IGNORE-ERRORS**.[4]

```
(defun file-exists-p (pathname)
  #+(or sbcl lispworks openmcl)
  (probe-file pathname)

  #+(or allegro cmu)
  (or (probe-file (pathname-as-directory pathname))
      (probe-file pathname))

  #+clisp
  (or (ignore-errors
        (probe-file (pathname-as-file pathname)))
      (ignore-errors
        (let ((directory-form (pathname-as-directory pathname)))
          (when (ext:probe-directory directory-form)
            directory-form))))

  #-(or sbcl cmu lispworks openmcl allegro clisp)
  (error "file-exists-p not implemented"))
```

The function `pathname-as-file` that you need for the CLISP implementation of `file-exists-p` is the inverse of the previously defined `pathname-as-directory`, returning a pathname that's the file form equivalent of its argument. This function, despite being needed here only by CLISP, is generally useful, so define it for all implementations and make it part of the library.

```
(defun pathname-as-file (name)
  (let ((pathname (pathname name)))
    (when (wild-pathname-p pathname)
      (error "Can't reliably convert wild pathnames."))
    (if (directory-pathname-p name)
      (let* ((directory (pathname-directory pathname))
             (name-and-type (pathname (first (last directory)))))
        (make-pathname
         :directory (butlast directory)
         :name (pathname-name name-and-type)
         :type (pathname-type name-and-type)
         :defaults pathname))
      pathname)))
```

Walking a Directory Tree

Finally, to round out this library, you can implement a function called `walk-directory`. Unlike the functions defined previously, this function doesn't need to do much of anything to smooth over implementation differences; it just needs to use the functions you've already defined. However, it's quite handy, and you'll use it several times in subsequent chapters. It will take the name of a directory and a function and call the function on the pathnames of all the files under the directory, recursively. It will also take two keyword arguments: `:directories` and `:test`. When `:directories` is true, it will call the function on the pathnames of directories as well as regular files. The `:test` argument, if provided, specifies another function that's invoked on each pathname before the main function is; the main function will be called only if the test function returns true.

```
(defun walk-directory (dirname fn &key directories (test (constantly t)))
  (labels
      ((walk (name)
         (cond
           ((directory-pathname-p name)
            (when (and directories (funcall test name))
              (funcall fn name))
            (dolist (x (list-directory name)) (walk x)))
           ((funcall test name) (funcall fn name)))))
    (walk (pathname-as-directory dirname))))
```

Now you have a useful library of functions for dealing with pathnames. As I mentioned, these functions will come in handy in later chapters, particularly Chapters 23 and 27, where you'll use `walk-directory` to crawl through directory trees containing spam messages and MP3 files.

But before we get to that, though, I need to talk about object orientation, the topic of the next two chapters.

---

[1]One slightly annoying consequence of the way read-time conditionalization works is that there's no easy way to write a fall-through case. For example, if you add support for another implementation to `foo` by adding another expression guarded with `#+`, you need to remember to also add the same feature to the `or` feature expression after the `#-` or the **`ERROR`** form will be evaluated after your new code runs.

[2]Another special value, `:wild-inferiors`, can appear as part of the directory component of a wild pathname, but you won't need it in this chapter.

[3]Implementations are allowed to return `:unspecific` instead of **`NIL`** as the value of pathname components in certain situations such as when the component isn't used by that implementation.

[4]This is slightly broken in the sense that if **`PROBE-FILE`** signals an error for some other reason, this code will interpret it incorrectly. Unfortunately, the CLISP documentation doesn't specify what errors might be signaled by **`PROBE-FILE`** and `probe-directory`, and experimentation seems to show that they signal `simple-file-error`s in most erroneous situations.