14. Files and File I/O

Common Lisp provides a rich library of functionality for dealing with files. In this chapter I'll focus on a few basic file-related tasks: reading and writing files and listing files in the file system. For these basic tasks, Common Lisp's I/O facilities are similar to those in other languages. Common Lisp provides a stream abstraction for reading and writing data and an abstraction, called *pathnames*, for manipulating filenames in an operating system-independent way. Additionally, Common Lisp provides other bits of functionality unique to Lisp such as the ability to read and write s-expressions.

Reading File Data

The most basic file I/O task is to read the contents of a file. You obtain a stream from which you can read a file's contents with the **`OPEN`** function. By default **`OPEN`** returns a character-based input stream you can pass to a variety of functions that read one or more characters of text: **`READ-CHAR`** reads a single character; **`READ-LINE`** reads a line of text, returning it as a string with the end-of-line character(s) removed; and **`READ`** reads a single s-expression, returning a Lisp object. When you're done with the stream, you can close it with the **`CLOSE`** function.

The only required argument to **`OPEN`** is the name of the file to read. As you'll see in the section "Filenames," Common Lisp provides a couple of ways to represent a filename, but the simplest is to use a string containing the name in the local file-naming syntax. So assuming that `/some/file/name.txt` is a file, you can open it like this:

```
(open "/some/file/name.txt")
```

You can use the object returned as the first argument to any of the read functions. For instance, to print the first line of the file, you can combine **`OPEN`**, **`READ-LINE`**, and **`CLOSE`** as follows:

```
(let ((in (open "/some/file/name.txt")))
  (format t "~a~%" (read-line in))
  (close in))
```

Of course, a number of things can go wrong while trying to open and read from a file. The file may not exist. Or you may unexpectedly hit the end of the file while reading. By default **`OPEN`** and the `READ-*` functions will signal an error in these situations. In Chapter 19, I'll discuss how to recover from such errors. For now, however, there's a lighter-weight solution: each of these functions accepts arguments that modify its behavior in these exceptional situations.

If you want to open a possibly nonexistent file without **OPEN** signaling an error, you can use the keyword argument `:if-does-not-exist` to specify a different behavior. The three possible values are `:error`, the default; `:create`, which tells it to go ahead and create the file and then proceed as if it had already existed; and **NIL**, which tells it to return **NIL** instead of a stream. Thus, you can change the previous example to deal with the possibility that the file may not exist.

```
(let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
  (when in
    (format t "~a~%" (read-line in))
    (close in)))
```

The reading functions--**READ-CHAR**, **READ-LINE**, and **READ**--all take an optional argument, which defaults to true, that specifies whether they should signal an error if they're called at the end of the file. If that argument is **NIL**, they instead return the value of their third argument, which defaults to **NIL**. Thus, you could print all the lines in a file like this:

```
(let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
  (when in
    (loop for line = (read-line in nil)
         while line do (format t "~a~%" line))
    (close in)))
```

Of the three text-reading functions, **READ** is unique to Lisp. This is the same function that provides the *R* in the REPL and that's used to read Lisp source code. Each time it's called, it reads a single s-expression, skipping whitespace and comments, and returns the Lisp object denoted by the s-expression. For instance, suppose `/some/file/name.txt` has the following contents:

```
(1 2 3)
456
"a string" ; this is a comment
((a b)
 (c d))
```

In other words, it contains four s-expressions: a list of numbers, a number, a string, and a list of lists. You can read those expressions like this:

```
CL-USER> (defparameter *s* (open "/some/file/name.txt"))
*S*
CL-USER> (read *s*)
(1 2 3)
CL-USER> (read *s*)
456
CL-USER> (read *s*)
"a string"
CL-USER> (read *s*)
((A B) (C D))
CL-USER> (close *s*)
T
```

As you saw in Chapter 3, you can use **PRINT** to print Lisp objects in "readable" form. Thus, whenever you need to store a bit of data in a file, **PRINT** and **READ** provide an easy way to do it without having to design a data format or write a parser. They even--as the previous example demonstrated--give you comments for free. And because s-expressions were designed to be human editable, it's also a fine format for things like configuration files.[1]

Reading Binary Data

By default **`OPEN`** returns character streams, which translate the underlying bytes to characters according to a particular character-encoding scheme.[2] To read the raw bytes, you need to pass **`OPEN`** an `:element-type` argument of `'(unsigned-byte 8)`.[3] You can pass the resulting stream to the function **`READ-BYTE`**, which will return an integer between 0 and 255 each time it's called. **`READ-BYTE`**, like the character-reading functions, also accepts optional arguments to specify whether it should signal an error if called at the end of the file and what value to return if not. In Chapter 24 you'll build a library that allows you to conveniently read structured binary data using **`READ-BYTE`**.[4]

Bulk Reads

One last reading function, **`READ-SEQUENCE`**, works with both character and binary streams. You pass it a sequence (typically a vector) and a stream, and it attempts to fill the sequence with data from the stream. It returns the index of the first element of the sequence that wasn't filled or the length of the sequence if it was able to completely fill it. You can also pass `:start` and `:end` keyword arguments to specify a subsequence that should be filled instead. The sequence argument must be a type that can hold elements of the stream's element type. Since most operating systems support some form of block I/O, **`READ-SEQUENCE`** is likely to be quite a bit more efficient than filling a sequence by repeatedly calling **`READ-BYTE`** or **`READ-CHAR`**.

File Output

To write data to a file, you need an output stream, which you obtain by calling **`OPEN`** with a `:direction` keyword argument of `:output`. When opening a file for output, **`OPEN`** assumes the file shouldn't already exist and will signal an error if it does. However, you can change that behavior with the `:if-exists` keyword argument. Passing the value `:supersede` tells **`OPEN`** to replace the existing file. Passing `:append` causes **`OPEN`** to open the existing file such that new data will be written at the end of the file, while `:overwrite` returns a stream that will overwrite existing data starting from the beginning of the file. And passing **`NIL`** will cause **`OPEN`** to return **`NIL`** instead of a stream if the file already exists. A typical use of **`OPEN`** for output looks like this:

```
(open "/some/file/name.txt" :direction :output :if-exists :supersede)
```

Common Lisp also provides several functions for writing data: **`WRITE-CHAR`** writes a single character to the stream. **`WRITE-LINE`** writes a string followed by a newline, which will be output as the appropriate end-of-line character or characters for the platform. Another function,

WRITE-STRING, writes a string without adding any end-of-line characters. Two different functions can print just a newline: **TERPRI**--short for "terminate print"--unconditionally prints a newline character, and **FRESH-LINE** prints a newline character unless the stream is at the beginning of a line. **FRESH-LINE** is handy when you want to avoid spurious blank lines in textual output generated by different functions called in sequence. For example, suppose you have one function that generates output that should always be followed by a line break and another that should start on a new line. But assume that if the functions are called one after the other, you don't want a blank line between the two bits of output. If you use **FRESH-LINE** at the beginning of the second function, its output will always start on a new line, but if it's called right after the first, it won't emit an extra line break.

Several functions output Lisp data as s-expressions: **PRINT** prints an s-expression preceded by an end-of-line and followed by a space. **PRIN1** prints just the s-expression. And the function **PPRINT** prints s-expressions like **PRINT** and **PRIN1** but using the "pretty printer," which tries to print its output in an aesthetically pleasing way.

However, not all objects can be printed in a form that **READ** will understand. The variable ***PRINT-READABLY*** controls what happens if you try to print such an object with **PRINT**, **PRIN1**, or **PPRINT**. When it's **NIL**, these functions will print the object in a special syntax that's guaranteed to cause **READ** to signal an error if it tries to read it; otherwise they will signal an error rather than print the object.

Another function, **PRINC**, also prints Lisp objects, but in a way designed for human consumption. For instance, **PRINC** prints strings without quotation marks. You can generate more elaborate text output with the incredibly flexible if somewhat arcane **FORMAT** function. I'll discuss some of the more important details of **FORMAT**, which essentially defines a mini-language for emitting formatted output, in Chapter 18.

To write binary data to a file, you have to **OPEN** the file with the same `:element-type` argument as you did to read it: `'(unsigned-byte 8)`. You can then write individual bytes to the stream with **WRITE-BYTE**.

The bulk output function **WRITE-SEQUENCE** accepts both binary and character streams as long as all the elements of the sequence are of an appropriate type for the stream, either characters or bytes. As with **READ-SEQUENCE**, this function is likely to be quite a bit more efficient than writing the elements of the sequence one at a time.

Closing Files

As anyone who has written code that deals with lots of files knows, it's important to close files when you're done with them, because file handles tend to be a scarce resource. If you open files and don't close them, you'll soon discover you can't open any more files.[5] It might seem straightforward enough to just be sure every **OPEN** has a matching **CLOSE**. For instance, you could always structure your file using code like this:

```
(let ((stream (open "/some/file/name.txt")))
  ;; do stuff with stream
  (close stream))
```

However, this approach suffers from two problems. One is simply that it's error prone--if you forget the **CLOSE**, the code will leak a file handle every time it runs. The other--and more significant--problem is that there's no guarantee you'll get to the **CLOSE**. For instance, if the code prior to the **CLOSE** contains a **RETURN** or **RETURN-FROM**, you could leave the **LET** without closing the stream. Or, as you'll see in Chapter 19, if any of the code before the **CLOSE** signals an error, control may jump out of the **LET** to an error handler and never come back to close the stream.

Common Lisp provides a general solution to the problem of how to ensure that certain code always runs: the special operator **UNWIND-PROTECT**, which I'll discuss in Chapter 20. However, because the pattern of opening a file, doing something with the resulting stream, and then closing the stream is so common, Common Lisp provides a macro, **WITH-OPEN-FILE**, built on top of **UNWIND-PROTECT**, to encapsulate this pattern. This is the basic form:

```
(with-open-file (stream-var open-argument*)
  body-form*)
```

The forms in *body-forms* are evaluated with *stream-var* bound to a file stream opened by a call to **OPEN** with *open-arguments* as its arguments. **WITH-OPEN-FILE** then ensures the stream in *stream-var* is closed before the **WITH-OPEN-FILE** form returns. Thus, you can write this to read a line from a file:

```
(with-open-file (stream "/some/file/name.txt")
  (format t "~a~%" (read-line stream)))
```

To create a new file, you can write something like this:

```
(with-open-file (stream "/some/file/name.txt" :direction :output)
  (format stream "Some text."))
```

You'll probably use **WITH-OPEN-FILE** for 90-99 percent of the file I/O you do--the only time you need to use raw **OPEN** and **CLOSE** calls is if you need to open a file in a function and keep the stream around after the function returns. In that case, you must take care to eventually close the stream yourself, or you'll leak file descriptors and may eventually end up unable to open any more files.

Filenames

So far you've used strings to represent filenames. However, using strings as filenames ties your code to a particular operating system and file system. Likewise, if you programmatically construct names according to the rules of a particular naming scheme (separating directories with /, say), you also tie your code to a particular file system.

To avoid this kind of nonportability, Common Lisp provides another representation of filenames: pathname objects. Pathnames represent filenames in a structured way that makes them easy to manipulate without tying them to a particular filename syntax. And the burden of translating back and forth between strings in the local syntax--called *namestrings*--and pathnames is placed on the Lisp implementation.

Unfortunately, as with many abstractions designed to hide the details of fundamentally different underlying systems, the pathname abstraction introduces its own complications. When pathnames were designed, the set of file systems in general use was quite a bit more variegated than those in common use today. Consequently, some nooks and crannies of the pathname abstraction make little sense if all you're concerned about is representing Unix or Windows filenames. However, once you understand which parts of the pathname abstraction you can ignore as artifacts of pathnames' evolutionary history, they do provide a convenient way to manipulate filenames.[6]

Most places a filename is called for, you can use either a namestring or a pathname. Which to use depends mostly on where the name originated. Filenames provided by the user--for example, as arguments or as values in configuration files--will typically be namestrings, since the user knows what operating system they're running on and shouldn't be expected to care about the details of how Lisp represents filenames. But programmatically generated filenames will be pathnames because you can create them portably. A stream returned by **`OPEN`** also represents a filename, namely, the filename that was originally used to open the stream. Together these three types are collectively referred to as *pathname designators*. All the built-in functions that expect a filename argument accept all three types of pathname designator. For instance, all the places in the previous section where you used a string to represent a filename, you could also have passed a pathname object or a stream.

How We Got Here

The historical diversity of file systems in existence during the 70s and 80s can be easy to forget. Kent Pitman, one of the principal technical editors of the Common Lisp standard, described the situation once in comp.lang.lisp (Message-ID: `sfwzo74np6w.fsf@world.std.com`) thusly:

> The dominant file systems at the time the design [of Common Lisp] was done were TOPS-10, TENEX, TOPS-20, VAX VMS, AT&T Unix, MIT Multics, MIT ITS, not to mention a bunch of mainframe [OSs]. Some were uppercase only, some mixed, some were case-sensitive but case- translating (like CL). Some had dirs as files, some not. Some had quote chars for funny file chars, some not. Some had wildcards, some didn't. Some had :up in relative pathnames, some didn't. Some had namable root dirs, some didn't. There were file systems with no directories, file systems with non-hierarchical directories, file systems with no file types, file systems with no versions, file systems with no devices, and so on.

If you look at the pathname abstraction from the point of view of any single file system, it seems baroque. However, if you take even two such similar file systems as Windows and Unix, you can already begin to see differences the pathname system can help abstract away--Windows filenames contain a drive letter, for instance, while Unix filenames don't. The other advantage of having the pathname abstraction designed to handle the wide variety of file systems that existed in the past is that it's more likely to be able to handle file systems that may exist in the future. If, say, versioning file systems come back into vogue, Common Lisp will be ready.

How Pathnames Represent Filenames

A pathname is a structured object that represents a filename using six components: host, device, directory, name, type, and version. Most of these components take on atomic values, usually strings; only the directory component is further structured, containing a list of directory names (as strings) prefaced with the keyword `:absolute` or `:relative`. However, not all pathname components are needed on all platforms--this is one of the reasons pathnames strike many new Lispers as gratuitously complex. On the other hand, you don't really need to worry about which components may or may not be used to represent names on a particular file system unless you need to create a new pathname object from scratch, which you'll almost never need to do. Instead, you'll usually get hold of pathname objects either by letting the implementation parse a file system-specific namestring into a pathname object or by creating a new pathname that takes most of its components from an existing pathname.

For instance, to translate a namestring to a pathname, you use the **`PATHNAME`** function. It takes a pathname designator and returns an equivalent pathname object. When the designator is already a pathname, it's simply returned. When it's a stream, the original filename is extracted and returned. When the designator is a namestring, however, it's parsed according to the local filename syntax. The language standard, as a platform-neutral document, doesn't specify any particular mapping from namestring to pathname, but most implementations follow the same conventions on a given operating system.

On Unix file systems, only the directory, name, and type components are typically used. On Windows, one more component--usually the device or host--holds the drive letter. On these platforms, a namestring is parsed by first splitting it into elements on the path separator--a slash on Unix and a slash or backslash on Windows. The drive letter on Windows will be placed into either the device or the host component. All but the last of the other name elements are placed in a list starting with `:absolute` or `:relative` depending on whether the name (ignoring the drive letter, if any) began with a path separator. This list becomes the directory component of the pathname. The last element is then split on the rightmost dot, if any, and the two parts put into the name and type components of the pathname.[7]

You can examine these individual components of a pathname with the functions **`PATHNAME-DIRECTORY`**, **`PATHNAME-NAME`**, and **`PATHNAME-TYPE`**.

```
(pathname-directory (pathname "/foo/bar/baz.txt")) ==> (:ABSOLUTE "foo" "bar")
(pathname-name (pathname "/foo/bar/baz.txt"))      ==> "baz"
(pathname-type (pathname "/foo/bar/baz.txt"))      ==> "txt"
```

Three other functions--**`PATHNAME-HOST`**, **`PATHNAME-DEVICE`**, and **`PATHNAME-VERSION`**--allow you to get at the other three pathname components, though they're unlikely to have interesting values on Unix. On Windows either **`PATHNAME-HOST`** or **`PATHNAME-DEVICE`** will return the drive letter.

Like many other built-in objects, pathnames have their own read syntax, `#p` followed by a double-quoted string. This allows you to print and read back s-expressions containing pathname objects, but because the syntax depends on the namestring parsing algorithm, such data isn't necessarily portable between operating systems.

```
(pathname "/foo/bar/baz.txt") ==> #p"/foo/bar/baz.txt"
```

To translate a pathname back to a namestring--for instance, to present to the user--you can use the function **`NAMESTRING`**, which takes a pathname designator and returns a namestring. Two other functions, **`DIRECTORY-NAMESTRING`** and **`FILE-NAMESTRING`**, return a partial namestring. **`DIRECTORY-NAMESTRING`** combines the elements of the directory component into a local directory name, and **`FILE-NAMESTRING`** combines the name and type components.[8]

```
(namestring #p"/foo/bar/baz.txt")           ==> "/foo/bar/baz.txt"
(directory-namestring #p"/foo/bar/baz.txt") ==> "/foo/bar/"
(file-namestring #p"/foo/bar/baz.txt")      ==> "baz.txt"
```

Constructing New Pathnames

You can construct arbitrary pathnames using the **`MAKE-PATHNAME`** function. It takes one keyword argument for each pathname component and returns a pathname with any supplied components filled in and the rest **`NIL`**.[9]

```
(make-pathname
  :directory '(:absolute "foo" "bar")
  :name "baz"
  :type "txt") ==> #p"/foo/bar/baz.txt"
```

However, if you want your programs to be portable, you probably don't want to make pathnames completely from scratch: even though the pathname abstraction protects you from unportable filename syntax, filenames can be unportable in other ways. For instance, the filename `/home/peter/foo.txt` is no good on an OS X box where `/home/` is called `/Users/`.

Another reason not to make pathnames completely from scratch is that different implementations use the pathname components slightly differently. For instance, as mentioned previously, some Windows-based Lisp implementations store the drive letter in the device component while others store it in the host component. If you write code like this:

```
(make-pathname :device "c" :directory '(:absolute "foo" "bar") :name "baz")
```

it will be correct on some implementations but not on others.

Rather than making names from scratch, you can build a new pathname based on an existing pathname with **MAKE-PATHNAME**'s keyword parameter `:defaults`. With this parameter you can provide a pathname designator, which will supply the values for any components not specified by other arguments. For example, the following expression creates a pathname with an `.html` extension and all other components the same as the pathname in the variable `input-file`:

```
(make-pathname :type "html" :defaults input-file)
```

Assuming the value in `input-file` was a user-provided name, this code will be robust in the face of operating system and implementation differences such as whether filenames have drive letters in them and where they're stored in a pathname if they do.[10]

You can use the same technique to create a pathname with a different directory component.

```
(make-pathname :directory '(:relative "backups") :defaults input-file)
```

However, this will create a pathname whose whole directory component is the relative directory `backups/`, regardless of any directory component `input-file` may have had. For example:

```
(make-pathname :directory '(:relative "backups")
               :defaults #p"/foo/bar/baz.txt") ==> #p"backups/baz.txt"
```

Sometimes, though, you want to combine two pathnames, at least one of which has a relative directory component, by combining their directory components. For instance, suppose you have a relative pathname such as `#p"foo/bar.html"` that you want to combine with an absolute pathname such as `#p"/www/html/"` to get `#p"/www/html/foo/bar.html"`. In that case, **MAKE-PATHNAME** won't do; instead, you want **MERGE-PATHNAMES**.

MERGE-PATHNAMES takes two pathnames and merges them, filling in any **NIL** components in the first pathname with the corresponding value from the second pathname, much like **MAKE-PATHNAME** fills in any unspecified components with components from the `:defaults` argument. However, **MERGE-PATHNAMES** treats the directory component specially: if the first pathname's directory is relative, the directory component of the resulting pathname will be the first pathname's directory relative to the second pathname's directory. Thus:

```
(merge-pathnames #p"foo/bar.html" #p"/www/html/") ==> #p"/www/html/foo/bar.html"
```

The second pathname can also be relative, in which case the resulting pathname will also be relative.

```
(merge-pathnames #p"foo/bar.html" #p"html/") ==> #p"html/foo/bar.html"
```

To reverse this process and obtain a filename relative to a particular root directory, you can use the handy function **ENOUGH-NAMESTRING**.

```
(enough-namestring #p"/www/html/foo/bar.html" #p"/www/") ==> "html/foo/bar.html"
```

You can then combine **ENOUGH-NAMESTRING** with **MERGE-PATHNAMES** to create a pathname representing the same name but in a different root.

```
(merge-pathnames
  (enough-namestring #p"/www/html/foo/bar/baz.html" #p"/www/")
  #p"/www-backups/") ==> #p"/www-backups/html/foo/bar/baz.html"
```

MERGE-PATHNAMES is also used internally by the standard functions that actually access files in the file system to fill in incomplete pathnames. For instance, suppose you make a pathname with just a name and a type.

```
(make-pathname :name "foo" :type "txt") ==> #p"foo.txt"
```

If you try to use this pathname as an argument to **OPEN**, the missing components, such as the directory, must be filled in before Lisp will be able to translate the pathname to an actual filename. Common Lisp will obtain values for the missing components by merging the given pathname with the value of the variable ***DEFAULT-PATHNAME-DEFAULTS***. The initial value of this variable is determined by the implementation but is usually a pathname with a directory component representing the directory where Lisp was started and appropriate values for the host and device components, if needed. If invoked with just one argument, **MERGE-PATHNAMES** will merge the argument with the value of ***DEFAULT-PATHNAME-DEFAULTS***. For instance, if ***DEFAULT-PATHNAME-DEFAULTS*** is `#p"/home/peter/"`, then you'd get the following:

```
(merge-pathnames #p"foo.txt") ==> #p"/home/peter/foo.txt"
```

Two Representations of Directory Names

When dealing with pathnames that name directories, you need to be aware of one wrinkle. Pathnames separate the directory and name components, but Unix and Windows consider directories just another kind of file. Thus, on those systems, every directory has two different pathname representations.

One representation, which I'll call *file form*, treats a directory like any other file and puts the last element of the namestring into the name and type components. The other representation, *directory form*, places all the elements of the name in the directory component, leaving the name and type components **NIL**. If `/foo/bar/` is a directory, then both of the following pathnames name it.

```
(make-pathname :directory '(:absolute "foo") :name "bar") ; file form
(make-pathname :directory '(:absolute "foo" "bar"))       ; directory form
```

When you create pathnames with **MAKE-PATHNAME**, you can control which form you get, but you need to be careful when dealing with namestrings. All current implementations create file form pathnames unless the namestring ends with a path separator. But you can't rely on user-supplied namestrings necessarily being in one form or another. For instance, suppose you've prompted the user for a directory to save a file in and they entered `"/home/peter"`. If you pass that value as the `:defaults` argument of **`MAKE-PATHNAME`** like this:

```
(make-pathname :name "foo" :type "txt" :defaults user-supplied-name)
```

you'll end up saving the file in `/home/foo.txt` rather than the intended `/home/peter/foo.txt` because the `"peter"` in the namestring will be placed in the name component when `user-supplied-name` is converted to a pathname. In the pathname portability library I'll discuss in the next chapter, you'll write a function called `pathname-as-directory` that converts a pathname to directory form. With that function you can reliably save the file in the directory indicated by the user.

```
(make-pathname
  :name "foo" :type "txt" :defaults (pathname-as-directory user-supplied-name))
```

Interacting with the File System

While the most common interaction with the file system is probably **`OPEN`**ing files for reading and writing, you'll also occasionally want to test whether a file exists, list the contents of a directory, delete and rename files, create directories, and get information about a file such as who owns it, when it was last modified, and its length. This is where the generality of the pathname abstraction begins to cause a bit of pain: because the language standard doesn't specify how functions that interact with the file system map to any specific file system, implementers are left with a fair bit of leeway.

That said, most of the functions that interact with the file system are still pretty straightforward. I'll discuss the standard functions here and point out the ones that suffer from nonportability between implementations. In the next chapter you'll develop a pathname portability library to smooth over some of those nonportability issues.

To test whether a file exists in the file system corresponding to a pathname designator--a pathname, namestring, or file stream--you can use the function **`PROBE-FILE`**. If the file named by the pathname designator exists, **`PROBE-FILE`** returns the file's *truename*, a pathname with any file system-level translations such as resolving symbolic links performed. Otherwise, it returns **`NIL`**. However, not all implementations support using this function to test whether a directory exists. Also, Common Lisp doesn't provide a portable way to test whether a given file that exists is a regular file or a directory. In the next chapter you'll wrap **`PROBE-FILE`** with a new function, `file-exists-p`, that can both test whether a directory exists and tell you whether a given name is the name of a file or directory.

Similarly, the standard function for listing files in the file system, **`DIRECTORY`**, works fine for simple cases, but the differences between implementations make it tricky to use portably. In the next chapter you'll define a `list-directory` function that smoothes over some of these differences.

DELETE-FILE and **RENAME-FILE** do what their names suggest. **DELETE-FILE** takes a pathname designator and deletes the named file, returning true if it succeeds. Otherwise it signals a **FILE-ERROR**.[11]

RENAME-FILE takes two pathname designators and renames the file named by the first name to the second name.

You can create directories with the function **ENSURE-DIRECTORIES-EXIST**. It takes a pathname designator and ensures that all the elements of the directory component exist and are directories, creating them as necessary. It returns the pathname it was passed, which makes it convenient to use inline.

```
(with-open-file (out (ensure-directories-exist name) :direction :output)
   ...
   )
```

Note that if you pass **ENSURE-DIRECTORIES-EXIST** a directory name, it should be in directory form, or the leaf directory won't be created.

The functions **FILE-WRITE-DATE** and **FILE-AUTHOR** both take a pathname designator. **FILE-WRITE-DATE** returns the time in number of seconds since midnight January 1, 1900, Greenwich mean time (GMT), that the file was last written, and **FILE-AUTHOR** returns, on Unix and Windows, the file owner.[12]

To find the length of a file, you can use the function **FILE-LENGTH**. For historical reasons **FILE-LENGTH** takes a stream as an argument rather than a pathname. In theory this allows **FILE-LENGTH** to return the length in terms of the element type of the stream. However, since on most present-day operating systems, the only information available about the length of a file, short of actually reading the whole file to measure it, is its length in bytes, that's what most implementations return, even when **FILE-LENGTH** is passed a character stream. However, the standard doesn't require this behavior, so for predictable results, the best way to get the length of a file is to use a binary stream.[13]

```
(with-open-file (in filename :element-type '(unsigned-byte 8))
  (file-length in))
```

A related function that also takes an open file stream as its argument is **FILE-POSITION**. When called with just a stream, this function returns the current position in the file--the number of elements that have been read from or written to the stream. When called with two arguments, the stream and a position designator, it sets the position of the stream to the designated position. The position designator must be the keyword `:start`, the keyword `:end`, or a non-negative integer. The two keywords set the position of the stream to the start or end of the file while an integer moves to the indicated position in the file. With a binary stream the position is simply a byte offset into the file. However, for character streams things are a bit more complicated because of character-encoding issues. Your best bet, if you need to jump around within a file of textual data, is to only ever pass, as a second argument to the two-argument version of **FILE-POSITION**, a value previously returned by the one-argument version of **FILE-POSITION** with the same stream argument.

Other Kinds of I/O

In addition to file streams, Common Lisp supports other kinds of streams, which can also be used with the various reading, writing, and printing I/O functions. For instance, you can read data from, or write data to, a string using **STRING-STREAM**s, which you can create with the functions **MAKE-STRING-INPUT-STREAM** and **MAKE-STRING-OUTPUT-STREAM**.

MAKE-STRING-INPUT-STREAM takes a string and optional start and end indices to bound the area of the string from which data should be read and returns a character stream that you can pass to any of the character-based input functions such as **READ-CHAR**, **READ-LINE**, or **READ**. For example, if you have a string containing a floating-point literal in Common Lisp's syntax, you can convert it to a float like this:

```
(let ((s (make-string-input-stream "1.23")))
  (unwind-protect (read s)
    (close s)))
```

Similarly, **MAKE-STRING-OUTPUT-STREAM** creates a stream you can use with **FORMAT**, **PRINT**, **WRITE-CHAR**, **WRITE-LINE**, and so on. It takes no arguments. Whatever you write, a string output stream will be accumulated into a string that can then be obtained with the function **GET-OUTPUT-STREAM-STRING**. Each time you call **GET-OUTPUT-STREAM-STRING**, the stream's internal string is cleared so you can reuse an existing string output stream.

However, you'll rarely use these functions directly, because the macros **WITH-INPUT-FROM-STRING** and **WITH-OUTPUT-TO-STRING** provide a more convenient interface. **WITH-INPUT-FROM-STRING** is similar to **WITH-OPEN-FILE**--it creates a string input stream from a given string and then executes the forms in its body with the stream bound to the variable you provide. For instance, instead of the **LET** form with the explicit **UNWIND-PROTECT**, you'd probably write this:

```
(with-input-from-string (s "1.23")
  (read s))
```

The **WITH-OUTPUT-TO-STRING** macro is similar: it binds a newly created string output stream to a variable you name and then executes its body. After all the body forms have been executed, **WITH-OUTPUT-TO-STRING** returns the value that would be returned by **GET-OUTPUT-STREAM-STRING**.

```
CL-USER> (with-output-to-string (out)
            (format out "hello, world ")
            (format out "~s" (list 1 2 3)))
"hello, world (1 2 3)"
```

The other kinds of streams defined in the language standard provide various kinds of stream "plumbing," allowing you to plug together streams in almost any configuration. A **`BROADCAST-STREAM`** is an output stream that sends any data written to it to a set of output streams provided as arguments to its constructor function, **`MAKE-BROADCAST-STREAM`**.[14] Conversely, a **`CONCATENATED-STREAM`** is an input stream that takes its input from a set of input streams, moving from stream to stream as it hits the end of each stream. **`CONCATENATED-STREAM`**s are constructed with the function **`MAKE-CONCATENATED-STREAM`**, which takes any number of input streams as arguments.

Two kinds of bidirectional streams that can plug together streams in a couple ways are **`TWO-WAY-STREAM`** and **`ECHO-STREAM`**. Their constructor functions, **`MAKE-TWO-WAY-STREAM`** and **`MAKE-ECHO-STREAM`**, both take two arguments, an input stream and an output stream, and return a stream of the appropriate type, which you can use with both input and output functions.

In a **`TWO-WAY-STREAM`** every read you perform will return data read from the underlying input stream, and every write will send data to the underlying output stream. An **`ECHO-STREAM`** works essentially the same way except that all the data read from the underlying input stream is also echoed to the output stream. Thus, the output stream of an **`ECHO-STREAM`** stream will contain a transcript of both sides of the conversation.

Using these five kinds of streams, you can build almost any topology of stream plumbing you want.

Finally, although the Common Lisp standard doesn't say anything about networking APIs, most implementations support socket programming and typically implement sockets as another kind of stream, so you can use all the regular I/O functions with them.[15]

Now you're ready to move on to building a library that smoothes over some of the differences between how the basic pathname functions behave in different Common Lisp implementations.

---

[1]Note, however, that while the Lisp reader knows how to skip comments, it completely skips them. Thus, if you use **`READ`** to read in a configuration file containing comments and then use **`PRINT`** to save changes to the data, you'll lose the comments.

[2]By default **`OPEN`** uses the default character encoding for the operating system, but it also accepts a keyword parameter, `:external-format`, that can pass implementation-defined values that specify a different encoding. Character streams also translate the platform-specific end-of-line sequence to the single character `#\Newline`.

[3]The type `(unsigned-byte 8)` indicates an 8-bit byte; Common Lisp "byte" types aren't a fixed size since Lisp has run at various times on architectures with byte sizes from 6 to 9 bits, to say nothing of the PDP-10, which had individually addressable variable-length bit fields of 1 to 36 bits.

[4]In general, a stream is either a character stream or a binary stream, so you can't mix calls to **READ-BYTE** and **READ-CHAR** or other character-based read functions. However, some implementations, such as Allegro, support so-called bivalent streams, which support both character and binary I/O.

[5]Some folks expect this wouldn't be a problem in a garbage-collected language such as Lisp. It is the case in most Lisp implementations that a stream that becomes garbage will automatically be closed. However, this isn't something to rely on--the problem is that garbage collectors usually run only when memory is low; they don't know about other scarce resources such as file handles. If there's plenty of memory available, it's easy to run out of file handles long before the garbage collector runs.

[6]Another reason the pathname system is considered somewhat baroque is because of the inclusion of *logical pathnames*. However, you can use the rest of the pathname system perfectly well without knowing anything more about logical pathnames than that you can safely ignore them. Briefly, logical pathnames allow Common Lisp programs to contain references to pathnames without naming specific files. Logical pathnames could then be mapped to specific locations in an actual file system when the program was installed by defining a "logical pathname translation" that translates logical pathnames matching certain wildcards to pathnames representing files in the file system, so-called physical pathnames. They have their uses in certain situations, but you can get pretty far without worrying about them.

[7]Many Unix-based implementations treat filenames whose last element starts with a dot and don't contain any other dots specially, putting the whole element, with the dot, in the name component and leaving the type component **NIL**.

```
(pathname-name (pathname "/foo/.emacs")) ==> ".emacs"
(pathname-type (pathname "/foo/.emacs")) ==> NIL
```

However, not all implementations follow this convention; some will create a pathname with "" as the name and `emacs` as the type.

[8]The name returned by **FILE-NAMESTRING** also includes the version component on file systems that use it.

[9]The host component may not default to **NIL**, but if not, it will be an opaque implementation-defined value.

[10]For absolutely maximum portability, you should really write this:

```
(make-pathname :type "html" :version :newest :defaults input-file)
```

Without the `:version` argument, on a file system with built-in versioning, the output pathname would inherit its version number from the input file which isn't likely to be right--if the input file has been saved many times it will have a much higher version number than the generated HTML file. On implementations without file versioning, the `:version` argument should be ignored. It's up to you if you care that much about portability.

[11]See Chapter 19 for more on handling errors.

[12]For applications that need access to other file attributes on a particular operating system or file system, libraries provide bindings to underlying C system calls. The Osicat library at `http://common-lisp.net/project/osicat/` provides a simple API built using the Universal Foreign Function Interface (UFFI), which should run on most Common Lisps that run on a POSIX operating system.

[13]The number of bytes and characters in a file can differ even if you're not using a multibyte character encoding. Because character streams also translate platform-specific line endings to a single `#\Newline` character, on Windows (which uses CRLF as its line ending) the number of characters will typically be smaller than the number of bytes. If you really have to know the number of characters in a file, you have to bite the bullet and write something like this:

```
(with-open-file (in filename)
  (loop while (read-char in nil) count t))
```

or maybe something more efficient like this:

```
(with-open-file (in filename)
  (let ((scratch (make-string 4096)))
    (loop for read = (read-sequence scratch in)
          while (plusp read) sum read)))
```

[14]**`MAKE-BROADCAST-STREAM`** can make a data black hole by calling it with no arguments.

[15]The biggest missing piece in Common Lisp's standard I/O facilities is a way for users to define new stream classes. There are, however, two de facto standards for user-defined streams. During the Common Lisp standardization, David Gray of Texas Instruments wrote a draft proposal for an API to allow users to define new stream classes. Unfortunately, there wasn't time to work out all the issues raised by his draft to include it in the language standard. However, many implementations support some form of so-called Gray Streams, basing their API on Gray's draft proposal. Another, newer API, called Simple Streams, has been developed by Franz and included in Allegro Common Lisp. It was designed to improve the performance of user-defined streams relative to Gray Streams and has been adopted by some of the open-source Common Lisp implementations.