Pathnames Quick-Start & Quick-Reference

Gene Michael Stover

created Sunday, 2002 September 1
updated Wednesday, 2005 February 2

Contents

Chapter 1

Introduction

The Steele book ([Ste90,)] & the Common Lisp Hyperspec ([X3J,)] contain reference material, but pathnames look complex enough that the prospect of getting your head wrapped around them might be intimidating.

It may be complex in its details, but Common Lisp's file system interface is simple to use in practice. Pathnames & the file system interface aren't difficult to use, but it'd be handy to have an alternative discussion of them in a format that's appropriate for people to display in a browser window next to their edit windows while they write file-handling Lisp expressions. That's what I've attempted to capture here.

Chapter 2

New Stuff

2.1 Logical Pathnames & Current Directories

Need to do some research on this to verify it, then add the results to the rest of the document.

Lisps don't always take the operating system's notion of current directory into account in the same way when they convert logical pathnames to native pathnames.

The cases I've seen are *clisp* & SBCL. When *clisp* converts a logical pathname to a native pathname, the native pathname can be relative to the operating system's current directory, but SBCL always converts to an absolute native pathname.

2.2 block of new text

I need to integrate the following with the rest of this article.

A really good article that has a lot to do with pathnames is "Fight the System"[1], by Henrik Motakef. It's about `defsystem`, not exactly about pathnames, but the topics overlap, & the article talks about conventions in logical pathname hostnames. As of 28 May 2003, it's unfinished, but it's still really good. I can't recommend it fervently enough.

When one Lisp program needs to load another, it looks like a good convention might be to use the logical hostname CL-LIBRARY & then a string of logical directories to mimick the library's originator & enclosing packages. For example, if a library wants to load the Acme company's network performance library, it might use the logical pathname CL-LIBRARY:ACME;NETWORK;PERFORMANCE.LISP.

This resembles the package naming system recommended by Java.[2] I haven't

[1] http://www.henrik-motakef.de/defsystem.html

[2] At least it was recommended by Java in about 1997. I haven't kept up-to-date with that language of weekly paradigm changes & endless remora technologies, so it might not still be the Java convention.

seen this convention mentioned in Lisp literature, but it would make sense to me.

The installation instructions for a library should list all the logical pathnames the library uses to load other Lisp packages. It should provide suggestions & examples for creating the logical-to-true pathname maps.

Most importantly, a library should recommend the logical pathnames that should be used to load it. Authors of other software which relies on the library should use those recommended logical pathnames when they load the library. I believe that recommending pathnames & following those recommendations are as important as any other factor in making Lisp libraries that are conveniently & realistically re-usable by other Lisp software.

Chapter 3

Heuristics

Here's a list of the rules of thumb I recommend with links to the rationales.

1. Namestrings obtained from users are literals. Use them as-is to operate on files, or convert them to pathnames with `parse-namestring`. (Section 6)
2. Do not hard-code a full pathname unless it's a logical pathname.
3. When you hard-code components of a non-logical pathname, use local case, but remember that some file systems are not case-sensitive. (Section 7.1)

Chapter 4

Concepts

Instances of the Pathnames class identify sets of files. A pathname might identify (or locate) exactly one file, multiple files, or no file. Same goes for namestrings, which hold about the same information that pathnames do, but in a string, whereas a pathname is an object with attributes.

Pathnames also have a portable, platform-independant programmatic interface. Some examples of the functions in that API are the `pathname`, `make-pathname`, `pathname-name`, & `merge-pathnames` functions.

Though pathnames have a portable interface, their values are not portable. It's important to understand this. The programmatic interface to pathnames is portable, but their values aren't.

Because the values of pathnames are not portable, it is not safe to hard-code any namestring values unless they are *logical* namestrings.[1] It can be portable to hard-code some individual components of pathnames . . . if you're very, very careful.

That's where *logical* pathnames & namestrings come in. They have portable values, compliments of a set of *logical pathname translations* which are set during the installation & configuration process.

[1]The documentation calls them "logical pathname namestrings", but I think the "pathname" in that term is reduntant or even conflicting, so I prefer to leave it out. Hope that doesn't bother anyone.

Chapter 5

Before Pathnames

You can do a lot of file-system manipulation without any explicit use of pathnames because the file-system interface functions, such as `open` & `probe-file`, can operate on strings. Such a string is called a *namestring*.

So when your program reads a namestring as input & just needs to open the file, it can use the namestring without any concern for pathnames or related functions.

You only need to deal with pathnames when you parse namestrings or manipulate pathnames. The point of pathnames is to provide an interface for manipulating files that is independant of the file system's type. Logical pathname namestrings also provide an interface that is independent of the actual file system.

Chapter 6

Where Pathnames Come From

A pathname is an object that a Lisp program can manipulate internally. It's useful to think about how pathnames get into a program.

One way is for a user or other external entity to give the program a string, which would be a namestring. Users always enter platform-specific namestrings; they don't care about Lisp's notion of common case or logical pathnames.[1] To convert a user-supplied namestring to a pathname, a program should use `parse-namestring`.

When your program prints a pathname for a user, the user wants to see the pathname expressed in the conventions of the local system, so to convert a pathname to a string for the user to see, use the `namestring` function.

Another way for a pathname to get into a program is from the programmer. (That would be you.) The programmer does not know the details of the target platform, so she must enter them in a portable way. There are three main ways for a programmer to specify a pathname portably. They are:

1. Construct them from literal component values.[2] (Section 7.1)
2. Construct them from logical component values.
3. Use logical namestrings.

The third way for a pathname to get into a program is to generate it from one or more other pathnames by merging or altering them.

[1]If you do your job right, a user doesn't need to know your program is written in Lisp.
[2]"Literal component values" might not be a very good term for that. More thought is necessary.

Chapter 7

Portability Tips

7.1 Character Case

You'd think you could use "`:case :common`" to achieve portability where character case is concerned. You can, but it's easier & *no less portable* to use lower, local case ("`:case :local`"). Let me explain.

When you tell `make-pathname` & friends that a string is in local case, it must convert the values so that the local file system can deal with them[1], so lower-case values should work on a file system if no other reason prevents them from working.

Lower-case values are easier to type than the upper-case values required for customary case of common case, & you don't need to remember to use "`:case :common`" when you call functions to manipulate the values.

I'm pretty sure the the only time lower, local case fails is when a file system preserves case but it's customary case is upper. When both those conditions are true about a file system, & you hard-code a lower-case component, it'll still work technically, but if you print the pathname (after conversion to a namestring) for a user, it'll be lower-case, which might be a mild surprise to the user.

The only other gotcha about using lower-case pathname components is that a case-sensitive string comparisons could erroneously indicate a difference on file systems where the customary case is upper-case, but you shouldn't compare pathnames, namestrings, or their components as strings anyway. Compare them with `equal`; it will perform a comparison that is appropriate for the file system.

Here's why lower-case works in practice:

1. On a file system that supports only upper-case, the lower-case pathname will be converted to upper-case.

2. On a case-sensitive, case-preserving file system, the lower-case will be preserved, & most (all?) case-sensitive file systems prefer lower-case.

[1]Page 618 in [Ste90]. Section 19.2.2.1.2.1 of [X3J].

3. Finally, on a case-insensitive, case-preserving file system (Macintosh or MS-DOS 95 & later), the case will be preserved & won't conflict too much with the mixed-case conventions of those file systems.

Mixed-case can be safe if you are careful to ensure:

1. that all pathnames which should be `equal` have the same case (so it works correctly on case-sensitive file systems) &
2. that any two pathnames which should not be `equal` differ in some way other than just case (so it works correctly on file systems that ignore case).

If you can ensure those two rules, I *think* it's safe to use mixed-case.

7.1.1 Got an Example?

Yup.

You could do this:

```
> (make-pathname :name "DOWAH" :type "LISP" :case :common)
```

but I claim that it's easier, equivalent, & no less portable to do this:

```
> (make-pathname :name "dowah" :type "lisp")
```

Mixed case has the same result whether you use `:common` case or `:local` case, so you might as well use local case.

7.1.2 Why Does :common Exist?

Common case exists to convert from local case on one host, to a common case, & then to local case on another file system[2].

How often does that happen? Answer: Not bloody often. I can't even think of an example that isn't at least a little contrived. For starters, any pathname that has more components than just a name & a type is likely to be very non-portable. Lots of Unix file systems have a file called `/etc/passwd`, but it's a dying trend, & I presume that almost no MS-DOS or Macintosh systems have a file with that name. So what would it mean to move `/etc/passwd` to a Twenex or a Mac? Even if you are on another Unix system, do you have a `/home/gene` directory? And what if a pathname contains something seriously non-portable, like a host or a device? Those are unlikely to be portable even between file systems of the same type.

For translating names of files from one file system to another, `:common` case could be useful, but how often does that happen?

Remember that if you are hard-coding a file name, you don't need `:common` case at all. It won't buy you anything. If you hard-code a lower-case file-name, then you'd hard-code an upper-case filename (& remember the `:case`

[2]Page 617 in [Ste90].

`:common`). If you have a mixed-case string, you'd still have a mixed-case string with `:common` case. Hard-coding a filename is not the same as translating from one local representation to another.

I don't doubt that there are uses for `:common` case, but they must be few & far between.

7.2 Case & Namestrings from Users

When a user enters a namestring, it already uses the customary case of the file system. Do not force it to upper-case or lower-case. Don't do any other edit on it, for that matter. Parse it into a pathname or use it to open (or probe or copy or whatever) files as-is.

7.3 Miscellaneous

Use the `:newest` keyword for file versions whenever possible. Use it unless you know you need a specific version & you know that the file system supports versions. Prefer `:newest` to `:unspecific`.

Assume that Lisp source files have a type of `LISP`. (Fixme: This conflicts with the advice in (Section 7.1).)

Do not use a logical host name of `SYS`. Unless I need more than one logical host name (which is rare), I use a single logical hostname of `HOST`.

For maximum portability, the only pathnames or namestrings you should hard-code are logical pathnames or logical namestrings. Anything else reduces portability, though notice that there are a few cases in which I have not figured out how to do it without hard-coded a non-logical pathname or namestring.

7.4 Bad Examples

Don't do these things! These are examples of what *not* to do.

In particular, don't hard-code an expression such as "`(make-pathname :host "abacus" :device "C" :directory '(:absolute "My Documents" "werther-project") :name "wlskdjaeks" :type "doc")`". Don't be fooled into thinking that is portable because the `make-pathname` function is portable. In fact, this example demonstrates the difference between Common Lisp's portable programmatic interface to the file system & the non-portable nature of the values of pathnames & namestrings.

Also in that one example, notice the two ways in which the pathname's value can be computer-specific. First, it can depend on the file system's type. That expression can probably be processed just fine by any Common Lisp implementation on MS-DOS. Any such implementation will probably produce the pathname equivalent of "`//ABACUS/C$/My Documents/werther-project/wlskdjaeks.doc`", but a Common Lisp on another type of file system might be confused, especially by the device name. The second un-portable feature of that pathname affects even other MS-DOS file systems because they might not be on a network with a host named ABACUS, & that host might not have a directory called `werther-projects` containing a file called `wlskdjaeks.doc`. There are two types of portability where file systems are concerned, that of file system's type & that of file system itself.

Chapter 8

Logical Pathnames

The idea behind logical pathnames is that your program can hard-code them, manipulate them within a virtual file system which you have designed to be convenient for your program's purposes, but they are translated to pathnames for the actual file system. The translations are created by the human (or program) which installs & configures your program.

You design the logical file system for the convenience of your program. For example, if you are writing a mail program, you might want to plan for installations in which incoming mail is stored on one host, outgoing mail is sent through another host, & user home directories are on yet another host. So you might put three logical host names in the logical file system for your program: MAIL, SMTP, & HOMES. Your program will still work just fine on an installation that doesn't use three different hosts for all those tasks. You've simply prepared for the three-host contingency.

When your program is installed, the installation tool might present its human operator with a list of the important features of the logical file system. Those features would include hosts, notable directories, & file types. The installation program might also include notes & suggestions about what translations need defining. The human operator would create the translations & save them.

When your program runs, it loads the translations. You've hard-coded some logical pathnames & logical namestrings, which your program translates to actual pathnames. It might also read literal namestrings from the user. It can manipulate the literal pathnames, merge them, construct new ones, whatever. If done carefully, it can be done portably.

Chapter 9

Common Tasks

9.1 Loading the Logical Pathname Translations

Your program must load the logical pathname translations. I know of two techniques: They are hard-coded into your program, or your program loads them from a file.

In both cases, the installation program collects the translations from its human operator, & then it writes the translations. The difference is only in the location where the installation program writes the translations.

9.1.1 Hard-Coded Translation In Your Program

The installation program might hard-code the translations into your Lisp program. This could work well if you use a short Lisp file to load the rest of your program. The installation program could write that loader file, & it could hard-code the translations into the beginning of the file. Figure 9.1 shows an example of a Lisp "loader" file like this.

9.1.2 Loading Translations from Another File

I still assume you're using a loader file to load the rest of your program. Instead of hard-coding the logical pathname translations into the loader file, the installation program could hard-code them into another file, & the loader file could load that translations file.

The problem with this technique is that the installation program must hard-code a local namestring to load the file of translations. It's a chicken-&-egg problem.

```
;;;
;;; The example logical file system has three hosts:
;;; MAIL, where e-mail is kept,
;;; USERS, where user home directories are kept, &
;;; HOST, where the rest of this Lisp program is kept.
;;;
;;; The translations here might be suitable for some
;;; Unix system.
;;;
(setf (logical-pathname-translations "MAIL")
      '(("*.*.*" "/var/mail/*")))
(setf (logical-pathname-translations "USERS")
      '(("*" "/home/*")))
(setf (logical-pathname-translations "HOST")
      '(("**;*.LISP" "/usr/local/lib/lisp/myprogram/**/*.lisp")))

;;; Load the other parts of this program.
(load (translate-logical-pathname "HOST:IO.LISP"))
(load (translate-logical-pathname "HOST:CLASSES;BOX.LISP"))
(load (translate-logical-pathname "HOST:CLASSES;LOGAN.LISP"))
```

Figure 9.1: Example of a Lisp source file with embedded, hard-coded logical pathname translations

9.2 Logical Pathname Miscellaneous Things

You can specify logical namestrings in upper case or lower case. They are always converted to upper case internally, then converted to the customary case of your file system when they are converted to literal pathnames. I prefer to write them in upper case because it helps me remember they are logical, not local, namestrings & because all upper case looks so klunky computerish, which I find I like as I get older.

Because logical namestrings are translated to upper case first, they & their translated local pathnames are effectively insensitive to case. In other words, don't rely on case in any way in the pathnames your program manipulates. The translated local pathnames might contain mixed case or have case-sensitive parts, but the parts derived from parts of the logical pathname cannot have mixed case or be case-sensitive.

9.3 Namestring to Pathname

If your program reads a literal, file-system-dependant namestring as an input & needs to manipulate it, you should convert it into a pathname so you can manipulate it in a way that is independant of the file-system's type. (That's pretty much the purpose of pathnames.)

When would this situation arise? Pretty much any time a user or some other external entity specifies a file for your Lisp program to manipulate. Whether you've prompted a user to enter a filename at the keyboard or displayed a graphical file-chooser window, your program will probably obtain a literal namestring that specifies a file according to the conventions of the file system.

The simplest way to convert a string to a pathname is with the `pathname` function. Just give it the string, & it'll give you a pathname. The namestring should contain only characters that are allowed in the file-system's pathnames. You should trim unwanted or padding characters from the string first.

For more control, you can use the `parse-namestring` function. The extra control mostly comes from the optional `:start`, `:end`, & `:junk-allowed` keyword arguments, but besides the pathname, `parse-namestring` also returns the number of characters parsed, in case you needed to parse the rest of the string. See [Ste90] or [X3J] for the details of `parse-namestring`.

9.4 Validating Namestring Input

9.5 Finding Your Program's Configuration File

9.5.1 Per-User

To find the configuration file for the current user, you program can call `user-homedir-pathname` without specifying the optional hostname argument. That function will return a pathname. If your program assumes a hard-coded name for the configuration file, you can use `merge-pathnames` to construct the pathname for your configuration file from the pathname you fetched from `user-homedir-pathname`. Figure 9.2 shows an example. It is also in per-user-config.lisp.

If your Lisp's `user-homedir-pathname` has the same notion of home directory as Unix does, this technique agrees well with the Unix way of doing things.

I don't know what Lisps for other operating systems consider a home directory, nor do I know what other operating systems consider a home directory, so I'm not sure how well this technique agrees with the conventions of those other environments.

Another potential problem with this technique is that you must choose a name & type for your configuration file that is expressable in any file system. I hope that a lower-case alphabetic word of 6 characters or less will work in all cases, with the Lisp converting the lower-case to upper-case (or whatever) if necessary. I know that a lower-case alphabetic word of 8 characters or less will work in most cases.[1]

9.5.2 System-Wide

A system-wide configuration file for a program is a configuration file that is loaded regardless of which user is running the program. An example from Unix might be `/etc/lynx.conf`, which is the configuration file for the `lynx` Web browser.

A task couldn't be easier. Just hard-code a logical namestring for your program's configuration file. The installation program is responsible emitting a translation to the actual configuration file.

If you have just one configuration file for the whole program, you might have a translation like the one in Figure 9.3.

To load that single configuration file, just EVAL "`(load (translate-logical-pathname "HOST:CONFIG.LISP"))`".

If your program's configuration is complex enough that it is separated into multiple files, you'll need a more complex translation, such as the one in Figure 9.4.

You'll need to translate a logical pathname & load the result for each of your configuration files. For example, you might need to translate, then load, `HOST:CONFIG;USERS.LISP`, then `HOST:CONFIG;PERMISSIONS.LISP`.

9.6 Computing One Pathname from Another

When your program reads a namestring as input, maybe from a human user who selects a file to open & process, you might need to construct a pathname from the namestring. Maybe your program is a text editor & tries to make a backup of a file before editing it. Maybe the user specifies a directory & a basename

[1]Let me be clear about "lower-case alphabetic" words. I mean a string containing only the letters *a* through *z*. No upper-case letters, no digits, no punctuation, no spaces, & no special characters.

```
;;; Here are some expressions which determine
;;; the pathname of "this" program's
;;; configuration file.  For clarity, I'll
;;; make liberal use of temporary variables,
;;; though that is not the usual Lisp style.

(defvar *home* (user-homedir-pathname)
  "User's home directory.  We'll borrow the
Unix convention that configuration files
live in a user's home directory.")

(defvar *name* "myprog"
  "The name of the configuration file.
It might be a good idea if it has the
same name as the program.  If we stuck
strictly with the Unix convention, it would
begin with a dot, but that isn't portable to
all file systems, \& it might give the
pathname functions of some Lisps a
heart-attack.")

(defvar *type* "rc"
  "The type of the configuration file.
Could be nearly anything.  For
portability, it's safest to have a file
type, \& there is precedent for rc.")

(defvar *basepath*
        (make-pathname :name *name*
                       :type *type*
                       :version :newest)
  "We'll create the config file's actual
pathname by merging this with the
pathname of the user's home directory.")

(defvar *config*
        (merge-pathnames *basepath* *home*)
  "And now *CONFIG* is bound to the
pathname of the config file.")

(format t "~&~A" (namestring *config*))

;;; --- end of file ---
```

Figure 9.2: Example of finding a per-user configuration file

```
;;;
;;; A logical pathname translation for a program
;;; with a single system-wide configuration file.
;;;
;;; The local namestring here is for Unix.
;;;
(setf (logical-pathname-translations "HOST")
      '("CONFIG.LISP" "/etc/myprogram-config.lisp"))
```

Figure 9.3: A logical pathname translation for a program with a single system-wide configuration file

```
;;;
;;; A logical pathname translation for a program
;;; with multiple system-wide configuration files.
;;;
;;; The local namestring here is for Unix.
;;;
(setf (logical-pathname-translations "HOST")
      '("CONFIG;*.LISP" "/etc/myprogram-config/*.lisp"))
```

Figure 9.4: A logical pathname translation for a program with multiple system-wide configuration file

The contents of merge-examples.lisp go here.

Figure 9.5: Some examples of merging pathnames

for your program, & your program creates several output files in that directory & with that name but with different types (a.k.a. filename extensions).

It's easy to construct pathnames from other pathnames for those examples. First, obtain a pathname from the namestring you read. Then call `merge-pathnames` using the first argument as a mask for the pathname you obtained from the user. Figure 9.5 has some examples of such mergest, & the Lisp source file containing them is online (merge-examples.lisp).

A problem with the examples in Figure 9.5 is that they hard-code the components of non-logical pathnames. That is technically non-portable, though I suspect it will work in the majority of modern cases. Nevertheless, it would be nice to know how to do equivalent translations using logical pathnames.

A complexity (problem) with obtaining the template (first) argument of `merge-pathnames` is that logical pathnames require logical hostnames, but the actual file system might not use hostnames (which is not a problem) or may use hostnames (which poses a problem).

fixme: How do you get a component for your pathname template that you give as the first argument to `merge-pathnames`? If you hard-code a literal, your program is not portable, but to use a logical namestring, you must provide a logical host, which will be portable only to homogenous networks. *???*

9.7 Obtaining a Portable Namestring from a Pathname

You can't.

You might think you could use the `namestring` function on a pathname with "`:case :common`" to get a portable string representing the pathname. Not so. That's because the string representation will probably contain your file system's idea of directory-name separators, host indicators (if it has one), type-separators, & version information (if it has that). The "`:case :common`" will put the string into a special representation that's designed to make case portable, but the string as a whole will not be portable.

I suppose that you might be able to search the logical pathnames you program hard-codes to find a representation that best fits a truename, but you'd have to implement it yourself. It could be a lot of work, & I could imagine situations in which there would be no unique result or not result at all.

Chapter 10

Example Code

10.1 The Current Directory

Many operating systems[1] have a concept of a current directory. In unix, it is accessed with the `getcwd` function in the C library & the `chdir` system call.

Common Lisp has a similar notion, but it is (or can be) implemented in the Lisp system, above the operating system, which is kind of cool in my opinion.

In Lisp, the current directory is bound to a global symbol called *DEFAULT-PATHNAME-DEFAULTS*.

When you create a pathname object with MAKE-PATHNAME, the attributes you don't specify are taken from the attributes in *DEFAULT-PATHNAME-DEFAULTS*, so *DEFAULT-PATHNAME-DEFAULTS* is basically your current directory. You can change Lisp's current directory by binding a different pathname to *DEFAULT-PATHNAME-DEFAULTS*.

A Lisp implementation may or may not make use the the underlying operating system's notion of a current directory. For portability, I would recommend that you assume Lisp initializes *DEFAULT-PATHNAME-DEFAULTS* from the operating system's current directory when you start Lisp, so if you start Lisp from the directory containing files you want to access, you may access those files without worrying about current or default directories. To access anything else, you should use a full pathname or change *DEFAULT-PATHNAME-DEFAULTS*.

10.1.1 Example

Here's an example of accessing files in different directories from within Lisp. The Lisp I used was SBCL 0.8.8. To show how its idea of the current directory interacts with the operating system's (some distro of Linux in this case, I

[1]I think I've used just three operating systems that didn't have a concept of a current directory. They were HP 2000, HP 3000, & IBM 360. On the other hand, all three had a concept of a place where a user kept his files; maybe that place could be considred a current directory. If so, then the current directory was set when the user logged in & could not be changed.

forget exactly which one), I'll do some of the work outside of Lisp, on the unix command line.

Let's start in a disposable temporary directory.

```
$ pwd
/home/gene/tmp/example
```

Inside that directory, let's create one data file. Let's also create a nested directory which contains yet another data file. I'll create those data files with unix's ECHO program just to be quick about it.[2]

```
$ echo \"this file is aa.lisp\" >aa.lisp
$ mkdir dir
$ echo \(choo choo cha cha\) >dir/bb.lisp
$
```

Okay, so now we have a file called `aa.lisp`, a directory called `dir`, & another file called `dir/bb.lisp`. Here's a picture of that in outline form in case that's easier to understand:

- `aa.lisp`
- `dir/`
  - `bb.lisp`

All of those are relative to the operating system's idea of the current directory.

Let's start SBCL from this current directory & see how we can access those files.

```
$ sbcl
This is SBCL 0.8.8, an implementation of ANSI Common Lisp.

More information about SBCL is available at <http://www.sbcl.org/>.
SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
*
```

First, take a look at *DEFAULT-PATHNAME-DEFAULTS*.

```
* *default-pathname-defaults*

#P"/home/gene/tmp/example/"
*
```

[2] I mention that because, if you aren't ready for it, seeing `echo` & a couple of escaped parents & an output redirection might be a bit of a shock.

After starting Lisp, the value bound to *DEFAULT-PATHNAME-DEFAULTS* is the same as the operating system's current directory. That means that if we use Lisp's DIRECTORY function without specifying the path parts of the pathname mask, we should see `aa.lisp` file & the `dir` directory.[3]

For convenience, let's make a pathname mask global variable.

```
* (defvar *pn* (make-pathname :name :wild :type :wild :version :wild))

*PN*
*
```

Now let's see what files we can see with that pathname mask. Notice that it does not specify any directory parts, so when we use the new mask, it will get its directory parts from *DEFAULT-PATHNAME-DEFAULTS*.

```
* (directory *pn*)

(#P"/home/gene/tmp/example/aa.lisp" #P"/home/gene/tmp/example/dir/")
*
```

So at this time, Lisp's idea of the current directory is the same as the operating system's when we started SBCL. The proof is in the pudding, so let's try reading from that `aa.lisp` file.

```
* (with-open-file (strm "aa.lisp")
    (read strm))

"this file is aa.lisp"
*
```

We opened a file called `aa.lisp`, & it was the same `aa.lisp` that we created from the unix command line.

Now let's do a similar thing with the `bb.lisp` file. It's in the directory `dir` which is in the current directory, so to access it, we will need to specify a directory part.

```
* (with-open-file (strm (make-pathname
                          :directory '(:relative "dir")
                          :name "bb" :type "lisp"
                          :version :newest))
    (read strm))

(CHOO CHOO CHA CHA)
*
```

[3]On some Lisps, such as `clisp`, you might not see the directories unless you take special steps. With the, you might see just files.

Now let's play with Lisp's idea of the current directory. We can do that by binding a new pathname object to *DEFAULT-PATHNAME-DEFAULTS*.

```
* (setq *default-pathname-defaults*
        (make-pathname :directory '(:relative "dir")))

#P"dir/"
* (directory *pn*)
WARNING:
   *DEFAULT-PATHNAME-DEFAULTS* is a relative pathname. (But we'll try using it
   anyway.)

NIL
*
```

Okay, so SBCL doesn't like a relative directory in *DEFAULT-PATHNAME-DEFAULTS*. Let's try something else. First, let's put *DEFAULT-PATHNAME-DEFAULTS* back the way it was.

```
* (setq *default-pathname-defaults* #P"/home/gene/tmp/example/")

#P"/home/gene/tmp/example/"
*
```

Let's construct a new *DEFAULT-PATHNAME-DEFAULTS* that is absolute. We can do this by mergine a new pathname with the current *DEFAULT-PATHNAME-DEFAULTS*.

```
* (setq *default-pathname-defaults*
        (merge-pathnames
          (make-pathname :directory '(:relative "dir"))
          *default-pathname-defaults*))

#P"/home/gene/tmp/example/dir/"
*
```

Now let's try the `directory` function again with the new *DEFAULT-PATHNAME-DEFAULTS*.

```
* (directory *pn*)

(#P"/home/gene/tmp/example/dir/bb.lisp")
*
```

Woohoo! Check it out. We've changed Lisp's idea of the current directory by binding a new pathname object to *DEFAULT-PATHNAME-DEFAULTS*. Remember that the new pathname needed to be absolute. Maybe some Lisp's won't

```
(defun wildcard-dir (pn)
  "PN is the pathname of a directory itself.  In other words,
(pathname-name pn) returns a string, not :WILD or another wildcard.
WILDCARD-DIR returns a new pathname that matches the names in
directory PN."
  (make-pathname :directory (append (pathname-directory pn)
                                    (list (pathname-name pn)))
                 :name :wild))

(defun traverse (root)
  (do ((x     (list root)
      (rest (append x (directory (wildcard-dir (first x))))))
       (count 0             (1+ count)))
      ((endp x) count)
    (format t "~&~A" (namestring (first x)))))
```

Figure 10.1: Lisp program to traverse directories

have a problem with relative directories in *DEFAULT-PATHNAME-DEFAULTS*, but SBCL wanted an absolute one, & we were able to make a new absolute pathname by merging a new pathname with the old value of *DEFAULT-PATHNAME-DEFAULTS*.

One last thing. Let's quit Lisp & see what the operating system says is our current directory now.

```
* (quit)
$ pwd
/home/gene/tmp/example
$
```

So changing *DEFAULT-PATHNAME-DEFAULTS* did not change the operating system's idea of the current directory (and I'm not surprised).

10.2 Directory Traversal

There's a directory-traversal program in Figure 10.1. It's available in a file by itself on the WWW at traverse.lisp.

You'd think that if I loaded that program & then evaluated on a Unix system "`(traverse (pathname ''/''))`", it would print every file & directory on the system, but no such luck. At least, no such luck with the clisp I use because its `directory` function doesn't include subdirectories in the list it returns. (See Section 11.1 for a discussion.)

If the behaviour of clisp's `directory` isn't a quirk, if it's true of other Lisps, then maybe you can't visits all the nodes in a subdirectory tree using plain Common Lisp.

Chapter 11

Quirks & Gotchas

11.1 clisp's directory function

clisp's `directory` function never includes other directories in the list it returns. Whether this is an implementation choice or a bug, I don't know. Specifically, it's clisp version 2.28 (released 2002-03-03).

I haven't tried the `directory` function on other Lisps. Don't know how they handle this situation.

Chapter 12

The Value of Portability

Some people might ask why anyone would even think about obscure file system with limits that are restrictive by today's standards, since systems like that are so rare as to be effectively non-existent. For example, file systems with 6-character limits on names or 3-character limits on types.[1]

Those people miss the point of portability. I used at least three file systems in college that had 6-char limits.[2] You may think that you will never use such a system, & you're probably right, which is too bad for you, 'cause alien computer systems from another age are fun, & they expand your mind, just like Lisp does.

The point of portability isn't just to help you port your programs to systems on which you predict you'll someday use them.[3] It's also so more people can port your programs to their alien computers.

Besides, if a *little* extra thought (which is what I'm recording in this paper) increases portability *hugely*, why not? I don't suggest that anyone spend years of their lives figuring out how to write a file system compatibility layer for computers with drum memory, or an OpenGL API for Babbage engines. Obviously, there are trade-offs & time limits. I'm talking about cases in which a reasonable amount of forethought & care produces a large pay-off in portability.

It is important, & it's fun. If you don't know that, you haven't been around long enough to port enough of your old programs.

[1]Someone from the past, using one of those systems, might wonder how anyone could use a file system that was so primitive that it did not support built-in versioning or complex record types.

[2]If I remember correctly, an HP/2000, an HP/3000, & a Cray something-or-other. I think the HPs ran something called MultiProgramming Environment (MPE), but I could be woefully mistaken. All of those file systems limited filenames to six characters. I think that at least one of them (can't remember which) didn't even have filename extensions (types).

[3]And how accurate can your predictions be?

Chapter 13

Blue Sky Dreaming

Notice that Common Lisp's logical pathnames support versioned files. The file systems of some operating systems which are no longer in the vogue supported versioned files, too. Modern, popular file systems do not support versioned files, though we have version control systems available for our modern operating systems. Examples of version systems include RCS, CVS, & Clearcase.

Wouldn't it be cool, or at least interesting, if a Lisp expanded on the local file system's namestring notation to include versioning, & it implemented that versioning by transparently making use of RCS, CVS, Clearcase, or some other versioning system installed on the modern operating system?

Chapter 14

Other Documentation

Of course, the definitive documentation is the Lisp standard, which does not appear to be online.

Some nearly definitive documentation is in the Steele book ([Ste90]) & the Common Lisp Hyperspec ([X3J]).

Chapter 15

Other Formats of this Document

Other formats of this document are available for your printing & reading enjoyment.

There is a PostScript version[1].

There is a Device Independent (DVI) version[2] which was the direct output of the LaTeX compiler.

I would make the Pointless Document Format (PDF) file available but `pdflatex` doesn't grok the LaTeX `html` package, & I don't have the time to beat it into cooperation. Sorry (seriously).

[1]pathnames-0.ps

[2]pathnames-0.dvi

Chapter 16

Unresolved Thoughts

These are thoughts & notes for my own benefit. (If I had answers to them or was certain they weren't off the deep end, I'd probably incorporate them into the rest of the article.)

- How to find libraries & other parts of the application at load-time? Logical pathnames?
- How to locate libraries at configuration & installation time? Let `./configure` do it? How to convey this information to the application? Does `./configure` write a "loader" file that contains hard-coded logical pathname translations & then evaluates `load` for all the files it needs? But if you were writing a custom loader file, why bother with logical pathname translations at all? Why not write actual, system-specific pathnames into the loader file? If you put the logical pathname translations into a file of their own (produced by `./configure`), you still need to hard-coded the literal pathname to that file in the loader file. I guess it's a chicken-&-egg problem, & it suggests that logical pathnames have little use.

Bibliography

[Ste90] Guy L. Steele. *Common Lisp: The Language*. Butterworth-Heinemann, second edition, 1990. ISBN 1-55558-041-6.

[X3J] ANSI Committee X3J13. Common lisp hyperspec. Xanalys Web site. http://www.lispworks.com/reference/HyperSpec/.