ANSI Common Lisp におけるプログラムの移植性

—— プラットフォームに依存しないファイルの指定方法 ——

知識工学部　工藤 千加子

2008 年 10 月 31 日

目 次

1 はじめに

プログラムの移植とは、既に実現されているプログラムを別の OS (プラットフォーム) や言語処理系で動くよう、プログラム修正することである。この修正箇所が少なければ少ないほど、移植性の高いプログラムであると言える。

ANSI Common Lisp は、プログラムの移植性を高めるためのファイルの指定方法が、言語仕様として定められている言語である。OS 間におけるファイルの指定方法の違いは、プログラミングによってではなく言語処理系によって吸収されるよう定められている。このため、ファイルの指定方法に関しては、OS 間のみならず、ANSI Common Lisp 準拠の言語処理系間においても、移植性の高いものとなる。

ANSI Common Lisp Spec Document[1] の 19.1 章には、次のように記述されている。

> 19.1 Overview of Filenames
>
> There are many kinds of file systems, varying widely both in their superficial syntactic details, and in their underlying power and structure. The facilities provided by Common Lisp for referring to and manipulating files has been chosen to be compatible with many kinds of file systems, while at the same time minimizing the program-visible differences between kinds of file systems.

このレポートでは、まず、ANSI Common Lisp Spec Document の 19 章の内容を説明することにより、どのようにしてプラットフォームに依存しないファイルの指定方法が可能になるのかを示す。詳しくは言語仕様を読むべきだと考えるため、参照すべき ANSI Common Lisp Spec Document の章番号も記しておく。

次に、ANSI Common Lisp 準拠の言語処理系である AllegroCL[2] における、ファイルの指定方法に関する固有の仕様を示す。

最後に、プラットフォームに依存しないファイルの指定方法を用い、OS 間と ANSI Common Lisp 準拠の処理系間における、プログラム移植の実例を示す。OS としては CentOS5, WindowsXP, Mac OS X(10.5.1) を用い、処理系としては AllegroCL と CMUCL[3] を用いる。

《このレポートにおける表記》

- 章参照について

項目	表記	例
ANSI Spec Document の章番号	イタリック字体で “*ANSI* 章番号”	*ANSI19.2* 章
本レポートの章番号	明朝字体で “章番号”	3 章

- 英字について

項目	表記	例
ANSI Spec Document で使用されている名詞	イタリック字体	*Pathname*
Lisp の関数名やクラス名	大文字タイプライター字体	`PATHNAME`

[1] ANSI Common Lisp http://franz.com/support/documentation/8.1/ansicl/ansicl.htm
[2] AllegroCL http://franz.com
[3] CMUCL http://www.cons.org/cmucl/

2 ファイルの指定

ANSI Common Lisp で規定されているファイルの指定方法は、*Pathname* による指定と *Namestring* による指定の 2 種類ある。*Namestring* は更に、*Logical Pathname Namestring* と *Physical Pathname Namestring* の 2 種類に分けられる (*ANSI19.1* 章参照)。

Pathname		host, device, directory, name, type, version の 6 つの構成要素からなる `PATHNAME` クラス *structure* object による指定
Namestring		文字列によりファイルを指定
	Logical Pathname Namestring	ANSI Common Lisp で規定された syntax に従った文字列
	Physical Pathname Namestring	*Logical Pathname Namestring* 以外の文字列。そのフォーマットは処理系に依存する。一般的に、“`/usr/local`” や “`C:\Program Files`” といった、OS に依存したファイル指定の文字列をサポートしている

移植性の高いアプリケーションを実現するためには、処理系に依存しない *Pathname* か *Logical Pathname Namestring* を使用すべきである。

全ての *Logical Pathname Namestring* は、`PATHNAME` を継承した `LOGICAL-PATHNAME` クラスの object に変換でき、全ての *Physical Pathname Namestring* もまた、処理系に依存したパーシング方法により `PATHNAME` クラスの object に変換できる。つまり、ANSI Common Lisp におけるファイルを指定する基本的な構造が、`PATHNAME` クラスであると言える。この構造で、OS 間におけるファイルの指定方法の違いを吸収している。

`PATHNAME` クラスの object の指すファイルが、実際にファイルシステム (OS) 上のどのファイルを指すかは、処理系に依存する。ファイルを特定できなかった場合、`FILE-ERROR` を発生させる (signaling する) ことが、ANSI Common Lisp で規定されている。ただし、この error 発生のタイミングは処理系依存と規定されているため、`PATHNAME` を生成した時かもしれないし、実際にそのファイルを `OPEN` しようとした時かもしれない (*ANSI19.1.2* 章参照)。

2.1 *Pathname*

Pathname は、PATHNAME クラス *structure* object を用いてファイルを指定する方法である。6 つの構成要素の詳細と指定できる値について、以下に示す。

構成要素	内容 (*ANSI19.2.1* 章)	指定できる値 (*ANSI19.2.2.4* 章)
host	ファイルが存在するファイルシステムの名前	処理系依存の object
device	ファイルが存在する device の名前	文字列, :WILD, :UNSPECIFIC, NIL
directory	ファイルが存在する directory の名前や階層構造	文字列, :WILD, :UNSPECIFIC, NIL, もしくは list 構造。list 構造は、directory 階層を表現する場合に用いられる。詳細は 2.1.1 章参照のこと。
name	ファイルの名前部分	文字列, :WILD, :UNSPECIFIC, NIL
type	ファイルのファイルタイプや拡張子	文字列, :WILD, :UNSPECIFIC, NIL
version	ファイルのバージョン番号	integer, :WILD, :UNSPECIFIC, :NEWEST, NIL。この他、処理系が定義した symbol。:NEWEST は一番大きな数字にマッチする。

指定できる値の特記事項を以下に示す。

値	項目	詳細
文字列	特殊文字の扱い (*ANSI19.2.2.1.1* 章)	“/” など、パスの区切り文字となる特殊文字は使用すべきではない。ただし、特殊文字を許すかどうかは処理系依存であり、特殊文字を許す場合は、処理系がファイルシステムに合わせて quote することになっている。
	大文字小文字の区別 (*ANSI19.2.2.1.2* 章)	大文字小文字の区別は、ファイルシステムにおけるその扱いに従う。例えば Windows はファイルや directory の名前に対して大文字と小文字の区別がないため、"foo" と "FOO" は同じ名前を意味するが、これを区別する UNIX では別の名前を意味する。
NIL	値の意味 (*ANSI19.2.2.2.1* 章)	値がないことを意味する。host 要素においては、処理系によっては default の値を意味する場合もある。
:WILD	値の意味 (*ANSI19.2.2.2.2* 章)	何にでも一致する wildcard を意味する。
:UNSPECIFIC	値の意味 (*ANSI19.2.2.2.3* 章)	意味を持たない構成要素であることを意味する。この値を各構成要素において使用できるかどうかは、処理系依存である。

PATHNAME の構成要素に NIL もしくは :UNSPECIFIC が設定されている object を *Namestring* に変換した場合、その構成要素に対する文字列は生成されない (*ANSI19.2.2.2.3.1* 章)。

2.1.1 directory 階層の表現

Directory の階層構造は、文字列と symbol から成る list で表現する。list の先頭 (car) は、以下の symbol でなければならない。

symbol	意味
:ABSOLUTE	root directory からのパス (=絶対パス) を意味する階層表現
:RELATIVE	default directory からの相対パスを意味する階層表現

list の 2 番目以降の値には、directory の名前を示す文字列、もしくは以下の symbol を指定できる。

symbol	意味
:WILD	directory 構造の 1 階層に一致
:WILD-INFERIORS	directory 構造のどの階層にも一致
:UP	directory 構造の 1 階層上に意味的に移動 (".." と同意義)
:BACK	directory 構造の 1 階層上に構文的に移動 (:BACK の前の文字列がないことと同意義)

:UP と :BACK の違いは、UNIX などのシンボリックリンクをサポートする OS において現れる。UNIX におけるこの違いの例を、directory の存在を確認する関数 PROBE-FILE を用いて示す。なお、この例の中で使用している "*" は、直前に実行した結果の第一戻り値を示す記号である (*ANSI25.2.22* 章参照)。

```
################################################################################
### directory 構成 （SHELL コマンドで、directory の存在を確認しておく）
###
$ ls -l /tmp/sample
total 8
lrwxrwxrwx 1 chika msi   12 Aug 14 18:11 bar -> /tmp/sample/foo/bar
drwxr-xr-x 2 chika msi 4096 Aug 14 18:12 baz
drwxr-xr-x 3 chika msi 4096 Aug 14 18:10 foo
$
$ ls -l /tmp/sample/bar/../baz/
ls: /tmp/sample/bar/../baz/: No such file or directory
$
$ ls -l /tmp/sample/baz/
total 0
-rw-r--r-- 1 chika msi 0 Aug 14 18:12 hello.cl
$

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; :up を使用した場合
;;;
;;;   ".." と同意義であるため、/tmp/sample/bar/../baz は存在しない。
;;;
CL-USER(10): (make-pathname :directory '(:absolute "tmp" "sample" "bar" :up "baz"))
#P"/tmp/sample/bar/../baz/"
CL-USER(11): (probe-file *)
NIL

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; :back を使用した場合
;;;
;;;   "bar" を指定していないのと同意義であるため、/tmp/sample/baz の存在を確認できる。
;;;
CL-USER(12): (make-pathname :directory '(:absolute "tmp" "sample" "bar" :back "baz"))
#P"/tmp/sample/baz/"
CL-USER(13): (probe-file *)
#P"/tmp/sample/baz/"
CL-USER(14):
```

2.2 *Logical Pathname Namestring*

Logical Pathname Namestring は、文字列でファイルを指定する方法である。その文字列の syntax は、ANSI Common Lisp で規定されている。以下にその syntax を示す。

```
=== Logical Pathname syntax ============================================================
logical-pathname::= [host host-marker]
                    [relative-directory-marker] {directory directory-marker}*
                    [name] [type-marker type [version-marker version]]

host::= word

directory::= word | wildcard-word | wild-inferiors-word

name::= word | wildcard-word

type::= word | wildcard-word

version::= pos-int | newest-word | wildcard-version

host-marker - a colon.

relative-directory-marker - a semicolon.

directory-marker - a semicolon.

type-marker - a dot.

version-marker - a dot.

wild-inferiors-word - The two character sequence "**" (two asterisks).

newest-word - The six character sequence "newest" or the six character sequence
              "NEWEST".

wildcard-version - an asterisk.

wildcard-word - one or more asterisks, uppercase letters, digits, and hyphens,
                including at least one asterisk, with no two asterisks adjacent.

word - one or more uppercase letters, digits, and hyphens.

pos-int - a positive integer.
=========================================================================================
```

この syntax における特記事項を、以下に示す (*ANSI19.2.1.1* 章参照)。

- host における予約語
  "SYS" は処理系のために用意されている予約語である。その解釈は処理系に依存する。
- PATHNAME に変換する時の device の値
  Logical Pathname Namestring には device を指定する syntax がない。PATHNAME に変換する際、常に :UNSPECIFIC となる。
- PATHNAME に変換する時の directory の値
  syntax 中の relative-directory-marker があれば :RELATIVE、なければ :ABSOLUTE となる。

この syntax の host について、説明を加えておく。

Logical Pathname から *Physical Pathname* への変換規則を、

```
(SETF (LOFICAL-PATHNAME-TRANSLATIONS host) new-translation)
```

で設定することができる。この引数の host を *logical host* と呼ぶ。*Logical host* を syntax の host に指定すると、*Logical Pathname Namestring* からその変換規則に従った *Physical Pathname Namestring* を得ることができる。以下に例をあげる。

```
CL-USER(6): (setf (logical-pathname-translations "MYPRJ")
                  '((";**;*.*" "/home/chika/prj/**/*.*")))
((";**;*.*" "/home/chika/prj/**/*.*"))
CL-USER(7):
CL-USER(7):  (translate-logical-pathname "MYPRJ:;foo;bar;baz.cl")
#P"/home/chika/prj/foo/bar/baz.cl"
CL-USER(8):
```

TRANSLATE-LOGICAL-PATHNAME 関数は、*Logical Pathname Namestring* を PATHNAME に変換する関数である。この関数が呼ばれた時、MYPRJ に対して設定した変換規則に従った PATHNAME が生成される。

この例は、UNIX の実行例であるため、パスの区切文字を “/” にしているが、区切文字の異なる OS や ファイル構成の異る OS へは、"/home/chika/prj/**/*.*" を変更するだけで移植可能なプログラムを作成することができる。

「プログラムの見え方の違いを最小限に」という、ANSI Common Lisp の目的が果たされている例である。

2.3 様々な関数や変数

Pathname や *Namestring* に関係する関数や変数値を以下にあげる。これらの関数の関数仕様や変数値の詳細は、*ANSI19.4* 章 を参照して頂きたい。

関数名	機能概要	参照先
pathname	*pathname designator* の示す PATHNAME を返す。	*ANSI19.4.3* 章
make-pathname	PATHNAME の構成要素の値を指定して、PATHNAME を作成する。	*ANSI19.4.4* 章
pathnamep	object が PATHNAME タイプかどうかチェックする。	*ANSI19.4.5* 章
pathname-host pathname-device pathname-directory pathname-name pathname-type pathname-version	*pathname designator* の PATHNAME としての各要素値を返す。	*ANSI19.4.6* 章
load-logial -pathname-translations	*logical host* の定義を load する。	*ANSI19.4.7* 章
logical-pathname -translations	*logical host* の定義を取得する。(setf (logical-pathname-translations host) translations) 関数により、*logical host* を定義することができる。	*ANSI19.4.8* 章
logical-pathname	*pathspec* を *Logical Pathname* に変換する。	*ANSI19.4.9* 章
default-pathname -defaults	MERGE-PATHNAMES などの関数で用いられる、default の PATHNAME object。	*ANSI19.4.10* 章
namestring file-namestring directory-namestring host-namestring enough-namestring	*pathname designator* を *namestring* に変換する。	*ANSI19.4.11* 章
parse-namestring	文字列や *pathname*、file を指す STREAM を PATHNAME に変換する。	*ANSI19.4.12* 章
wild-pathname-p	PATHNAME の構成要素の値に、wildcard が含まれているどうかチェックする。	*ANSI19.4.13* 章
pathname-match-p	*pathname designator* が *wildcard* にマッチするかどうかチェックする。	*ANSI19.4.14* 章
translate-logical -pathname	*pathname designator* や *logical pathname namestring* を *physical pathname* に変換する。	*ANSI19.4.15* 章
translate-pathname	変換規則を指定して、*pathname designator* を PATHNAME に変換する。	*ANSI19.4.16* 章
merge-pathnames	*pathname* をマージする。第一引数の *pathname* の不足構成要素を、第二引数 (もしくは *default-pathname-defaults*) の情報で埋める。	*ANSI19.4.17* 章

ANSI Common Lisp Spec Document に、これらの関数の様々な使用例が載っている。そちらも参照して欲しい。

3 AllegroCL における動作仕様

この章では、ファイルの指定方法に関する AllegroCL8.1 の動作仕様について、例をあげて説明する。ここであげる動作仕様に関しては、AllegroCL の PATHNAME に関するドキュメント[4] を参照すること。

3.1 UNIX 系における、シンボリックリンクの実体取得

UNIX はシンボリックリンクをサポートする OS である。Allegro CL の TRUENAME 関数では、ANSI Common Lisp で規定されている引数の他にキーワード引数 :FOLLOW-SYMLINKS が追加されており (default は T)、この値が T の時、リンク先の実体を取得することができる。

```
################################################################################
###OS: CentOS5
### directory 構成
###
$ ls -l /tmp/sample
total 8
lrwxrwxrwx 1 chika msi   12 Aug 14 18:11 bar -> /tmp/sample/foo/bar
drwxr-xr-x 2 chika msi 4096 Aug 14 18:12 baz
drwxr-xr-x 3 chika msi 4096 Aug 14 18:10 foo
$

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;OS: CentOS5
;;; truneme の引数による戻り値の違い
;;;
;;;  :follow-symlinks t の場合、link 先 (実体) のパスが戻り値となる。
;;;
CL-USER(9): (truename "/tmp/sample/bar/" :follow-symlinks nil)
#P"/tmp/sample/bar/"
CL-USER(10): (truename "/tmp/sample/bar/" :follow-symlinks t)
#P"/tmp/sample/foo/bar/"
CL-USER(11):
```

[4] http://franz.com/support/documentation/8.1/doc/pathnames.htm

3.2 Windows における、device の指定

Windows におけるファイルのパスには、c:\Program Files\ のように、device 名とコロンが含まれている (この例では "c" が device 名)。MAKE-PATHNAME で PATHNAME を生成するときは、この device 名を :DEVICE キーワード引数で指定しなければならない。

以下に、 c:\Program Files\acl81\alisp.exe の存在を確認する例をあげる。このファイルは、システム上に存在するファイルである。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;OS: WindowsXP
;;; Windows における PATHNAME の生成例
;;;
;;;   :device には "c" を指定すること。
;;;
CL-USER(2): (defvar *root* (make-pathname :device "c"
                                          :directory '(:ABSOLUTE "Program Files")))
*ROOT*
CL-USER(3): (describe *root*)
#P"c:\\Program Files\\" is a structure of type PATHNAME.  It has these
slots:
 HOST                NIL
 DEVICE              "c"
 DIRECTORY           (:ABSOLUTE "Program Files")
 NAME                NIL
 TYPE                NIL
 VERSION             :UNSPECIFIC
 NAMESTRING          "c:\\Program Files\\"
 HASH                NIL
 DIR-NAMESTRING      NIL
 PLIST               NIL
CL-USER(4): (setq p1 (merge-pathnames "acl81\\alisp.exe" *root*))
#P"c:\\Program Files\\acl81\\alisp.exe"
CL-USER(5): (probe-file p1)
#P"c:\\Program Files\\acl81\\alisp.exe"    ; ファイルの存在を確認できる。
CL-USER(6): (namestring p1)
"c:\\Program Files\\acl81\\alisp.exe"     ; p1 の文字列表記
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;OS: WindowsXP
;;; Windows における、間違った :device の使用例
;;;
;;;   p2 の :directory に device 名もセットされたため、
;;;   正しいパスとして表現できておらず、ファイルの存在を確認できない。
;;;
CL-USER(7): (defvar *acl-dir* "c:\\Program Files\\acl81\\")
*ACL-DIR*
CL-USER(8): (setq p2 (make-pathname :directory *acl-dir*
                                    :name "alisp"
                                    :type "exe"))
#P"c:\\Program Files\\acl81\\alisp.exe"
CL-USER(9): (namestring p2)
"c:\\Program Files\\acl81\\alisp.exe"      ; p2 の文字列表記は p1 と同じであるが、
CL-USER(10): (probe-file p2)               ; ファイルの存在を確認すると、
NIL                                        ; 確認できない。その理由は、
CL-USER(11): (describe p2)                 ; device 名が :directory に入っているため。
#P"c:\\Program Files\\acl81\\alisp.exe" is a structure of type
PATHNAME.  It has these slots:
 HOST               NIL
 DEVICE             NIL
 DIRECTORY          (:RELATIVE "c:" "Program Files" "acl81")
 NAME               "alisp"
 TYPE               "exe"
 VERSION            :UNSPECIFIC
 NAMESTRING         "c:\\Program Files\\acl81\\alisp.exe"
 HASH               NIL
 DIR-NAMESTRING     NIL
 PLIST              NIL
CL-USER(12):
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;OS: WindowsXP
;;; 先の間違った使用例を、PATHNAME 関数を利用して正しく参照できるよう修正した例
;;;
;;;    文字列は PATHNAME 関数を使用して処理系にパースさせ、MERGE-PATHNAMES 関数を使用する。
;;;    プログラマが PATHNAME の各要素を指定する必要はないため、スマートなコードになる。
;;;
CL-USER(12): (defvar *acl-pathname* (pathname *acl-dir*))
*ACL-PATHNAME*
CL-USER(13): (setq p3 (merge-pathnames "alisp.exe" *acl-pathname*))
#P"c:\\Program Files\\acl81\\alisp.exe"
CL-USER(14): (probe-file p3)              ; ファイルの存在を確認できる。
#P"c:\\Program Files\\acl81\\alisp.exe"
CL-USER(15): (describe p3)
#P"c:\\Program Files\\acl81\\alisp.exe" is a structure of type
PATHNAME.  It has these slots:
 HOST              NIL
 DEVICE            "c"
 DIRECTORY         (:ABSOLUTE "Program Files" "acl81")
 NAME              "alisp"
 TYPE              "exe"
 VERSION           :UNSPECIFIC
 NAMESTRING        "c:\\Program Files\\acl81\\alisp.exe"
 HASH              NIL
 DIR-NAMESTRING    NIL
 PLIST             NIL
CL-USER(16): (equal p1 p3)
T                                         ; p3 は p1 とは equal だが、
CL-USER(17): (equal p2 p3)                ; p2 とは equal ではない。
NIL
CL-USER(17):
```

3.3 UNIX における、文字列のパース

文字列を PATHNAME に変換するためのパース方法は、処理系に依存する。AllegroCL におけるパーシング規則は、AllegroCL PATHNAME に関するドキュメントの 3.0 章を参照して欲しい。ここでは、同章にもあがっている例と、その他、いくつかのパース例をあげる。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;OS: CentOS5
;;; 文字列から PATHNAME への変換例
;;;
CL-USER(4): (loop with formatter = "~15@< ~S~>~28@< ~S~>~10@< ~S~>~8@< ~S~>~%"
                initially (format t formatter 'arg 'directory 'name 'type)
                for pname in '("/" "/foo" "/foo." "/foo.b" "/foo.bar."
                               "/foo.bar.baz" "/foo/bar" "/foo..bar"
                               "foo.bar" "foo/" "foo/bar" "foo/bar/baz"
                               "foo/bar/" "foo/bar/.." "/foo/../"
                               ".lisprc" "x.lisprc" "." ".." "..." "~"
                               "/tmp/sample/bar/../baz/")
                as pathname = (pathname pname)
                do (format t formatter
                           pname
                           (pathname-directory pathname)
                           (pathname-name pathname)
                           (pathname-type pathname)))
 ARG            DIRECTORY                    NAME      TYPE
 "/"            (:ABSOLUTE)                  NIL       NIL
 "/foo"         (:ABSOLUTE)                  "foo"     NIL
 "/foo."        (:ABSOLUTE)                  "foo"     ""
 "/foo.b"       (:ABSOLUTE)                  "foo"     "b"
 "/foo.bar."    (:ABSOLUTE)                  "foo.bar" ""
 "/foo.bar.baz" (:ABSOLUTE)                  "foo.bar" "baz"
 "/foo/bar"     (:ABSOLUTE "foo")            "bar"     NIL
 "/foo..bar"    (:ABSOLUTE)                  "foo."    "bar"
 "foo.bar"      NIL                          "foo"     "bar"
 "foo/"         (:RELATIVE "foo")            NIL       NIL
 "foo/bar"      (:RELATIVE "foo")            "bar"     NIL
 "foo/bar/baz"  (:RELATIVE "foo" "bar")      "baz"     NIL
 "foo/bar/"     (:RELATIVE "foo" "bar")      NIL       NIL
 "foo/bar/.."   (:RELATIVE "foo")            NIL       NIL
 "/foo/../"     (:ABSOLUTE)                  NIL       NIL
 ".lisprc"      NIL                          ".lisprc" NIL
 "x.lisprc"     NIL                          "x"       "lisprc"
 "."            (:RELATIVE)                  NIL       NIL
 ".."           (:RELATIVE :BACK)            NIL       NIL
```

```
 "..."          NIL                          "..."     NIL
 "~"            (:ABSOLUTE "home" "chika")  NIL       NIL
 "/tmp/sample/bar/../baz/" (:ABSOLUTE "tmp" "sample" "baz") NIL       NIL
NIL
CL-USER(5):
```

3.4 Windows における、文字列のパース

Directory の区切文字は、“\\” と “/” の 2 種類である。以下に例をあげる。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;OS: WindowsXP
;;; 文字列から PATHNAME への変換例
;;;
;;;   3 つ目の例の "\\\\nara\\prj\\7\\" は、共有ファイルを指定する場合の例である。
;;;
CL-USER(3): (loop with formatter = "~22@< ~S~>~7@< ~S~>~30@< ~S~>~12@< ~S~>~8@< ~S~>~%"
               initially (format t formatter 'arg 'device 'directory 'name 'type)
               for pname in '("c:\\foo\\bar.exe" "c:/foo/bar/baz/"
                              "\\\\nara\\prj\\7\\")
               as pathname = (pathname pname)
               do (format t formatter
                          pname
                          (pathname-device pathname)
                          (pathname-directory pathname)
                          (pathname-name pathname)
                          (pathname-type pathname)))
 ARG                   DEVICE DIRECTORY                     NAME        TYPE
 "c:\\foo\\bar.exe"    "c"    (:ABSOLUTE "foo")             "bar"       "exe"
 "c:/foo/bar/baz/"     "c"    (:ABSOLUTE "foo" "bar" "baz") NIL         NIL
 "\\\\nara\\prj\\7\\"  "prj"  (:ABSOLUTE "7")               NIL         NIL
NIL
CL-USER(4):
```

3.4.1 *Logical Pathname* か *Physical Pathname* か

Logical Pathname Namestring の syntax において、host と directory の区切り文字にはコロンを使用することになっている。ここで、Windows における device と directory の区切り文字もコロンであるため、PATHNAME 関数を使用した場合、*Logical Pathname* として解釈されるのか *Physical Pathname* として解釈されるのかが、気になるところである。

これについて、AllegroCL PATHNAME のドキュメントには記載がない。いくつかの例を実行し、どのような規則なのかを確認しておく。

```
CL-USER(2): (loop with formatter
                  = "~18@< ~S~>~5@< ~S~>~7@< ~S~>~20@< ~S~>~12@< ~S~>~8@< ~S~>~%"
                  initially (format t formatter 'arg 'host 'device 'directory
                                    'name 'type)
                  for pname in '("c:foo.exe" "c:;foo.exe" "c:/foo.exe"
                                 "z:foo;bar.exe" "z:;foo;bar.exe" "z:/foo/bar.exe"
                                 "aa:foo.ext" "aa:foo;bar.exe"
                                 "aa:;foo;bar.exe" "aa:/foo/bar.exe")
                  as pathname = (pathname pname)
                  do (format t formatter
                             pname
                             (pathname-host pathname)
                             (pathname-device pathname)
                             (pathname-directory pathname)
                             (pathname-name pathname)
                             (pathname-type pathname)))
 ARG                HOST DEVICE DIRECTORY           NAME        TYPE
 "c:foo.exe"        NIL  "c"    NIL                 "foo"       "exe"
 "c:;foo.exe"       NIL  "c"    NIL                 ";foo"      "exe"
 "c:/foo.exe"       NIL  "c"    (:ABSOLUTE)         "foo"       "exe"
 "z:foo;bar.exe"    NIL  "z"    NIL                 "foo;bar"   "exe"
 "z:;foo;bar.exe"   NIL  "z"    NIL                 ";foo;bar"  "exe"
 "z:/foo/bar.exe"   NIL  "z"    (:ABSOLUTE "foo")   "bar"       "exe"
 "aa:foo.exe"       "aa" NIL    NIL                 "foo"       "exe"
 "aa:foo;bar.exe"   "aa" NIL    (:ABSOLUTE "foo")   "bar"       "exe"
 "aa:;foo;bar.exe"  "aa" NIL    (:RELATIVE "foo")   "bar"       "exe"
 "aa:/foo/bar.exe"  NIL  "aa"   (:ABSOLUTE "foo")   "bar"       "exe"
NIL
CL-USER(3):
```

この実行例から、コロンの前の文字列は、以下の規則で解釈されているようである。

1. コロンの前が英字 1 文字であれば、*Physical Pathname* の device と解釈。
2. それ以外で、コロン以降の文字が *Logical Pathname Namestring* の syntax の文字で構成されていれば、*Logical Pathname* の host と解釈。
3. それ以外は、*Physical Pathname* の device と解釈。

3.5 *Logical Pathname Namestring* に使用できる文字を追加

Logical Pathname Namestring の syntax で許されている記号は、ハイフンだけである。このため、“foo_sample.txt” といったファイル名は指定できないことになる。AllegroCL では、`EXCL:*ADDITIONAL-LOGICAL-PATHNAME-NAME-CHARS*` に `#\_` を追加することで、“_” を *Logiccal Pathname Namestring* として使うことができるよう、変更することができる。

この機能は一見便利そうに見えるが、処理系に依存しない *Logical Pathname Namestring* の syntax を壊して使用する方法であり、移植性を低くする。Syntax に合わないのであれば、`MAKE-PATHNAME` 関数や `PATHNAME` 関数を使用して `PATHNAME` を作成すべきであり、あえて syntax を壊してまで使うべき機能ではないと考える。このため、この機能に関する使用例は、このレポートでは提示しないことにした。

3.6 *Logical Pathname* の host における SYS の扱い

Logical Pathname Namestring の host で、“SYS” は予約であることを 2.2 章で述べた。AllegroCL における SYS は、AllegroCL 実行イメージが存在する directory を指す。

以下に例を示す。起動した Lisp のパスと SYS の指すパスが、同じであることを見て頂きたい。

```
#################################################################################
###OS: CentOS5
###
$ /usr/local/acl81/alisp
CL-USER(1): (translate-logical-pathname "SYS:")
#P"/usr/local/acl81/"
CL-USER(2):

#################################################################################
###OS: MacOS X
###
bash-3.2$ /Applications/AllegroCL/alisp
CL-USER(1): (translate-logical-pathname "SYS:")
#P"/Applications/AllegroCL/"
CL-USER(2):

#################################################################################
###OS: Windows
###  C:\Program Files\acl81\alisp.exe をクリックして起動
###
CL-USER(1): (translate-logical-pathname "SYS:")
#P"C:\\Program Files\\acl81\\"
CL-USER(2):

#################################################################################
###OS: CentOS5
### GENERATE-APPLICATION 関数により application 化した実行イメージの場合
###
$ /home/chika/work/application/reflector -qq
CL-USER(1): (translate-logical-pathname "SYS:")
#P"/home/chika/work/application/"
CL-USER(2):
```

AllegroCL におけるこの SYS は、最終的にどの directory に install されるかわからないアプリケーションを作成した場合にとても便利な文字列である。どこに install されても、SYS がその install directory を示すからである。

4　移植性をの高いアプリケーションを目指して

これまで述べてきたように、移植性の高いアプリケーションを作るためには、処理系依存となる機能をできるだけ使用しないことが重要である。PATHNAME で表現したファイルが実際にどのファイルを示すかは処理系に依存するが、directory を階層構造で管理している一般の OS においては、処理系毎に指すファイルが異なるとは考えにくい。よって、以下の点に気をつければ、移植性の高いアプリケーションを作成することができる。

1. 設計段階において、アプリケーションが出力するファイルや参照するファイルの名前は、できる限り *Logical Pathname Namestring* の syntax に従わせる。
2. コーディングの際、アプリケーション本体のメイン機能のプログラムでは、ファイル名を直接参照せずに、PATHNAME クラスの object を参照する。この PATHNAME クラスの object の定義はアプリケーション本体とは別に管理し、移植の時、この定義を変更すれば良いだけの形式にしておく。
3. 処理系に依存する関数や値は使用しない。

アプリケーションの開発において、最初から移植性を考慮する必要はない。必要になった時に、徐々に対応していけば良い。ただ、その対応が上にあげたちょっとした工夫で少なくなるのであれば、設計の段階から少しだけでも考慮すべきである。

以下の章では、簡単なアプリケーション開発を題材に、OS 間、処理系間のプログラム移植の例を示す。

4.1 アプリケーション例

以下の機能を満たすアプリケーションを作成する。

- このアプリケーションは、ある directory に存在するファイルの差分を、5 秒おきに監視するシステムである。
- 監視する directory は /usr/local/share とする。
- 前回チェック時との差分がある場合は、そのファイル名をログファイル〈実行 `directory`〉`/log/history.log` に出力する。
- 監視処理においてエラーが発生した場合、そのエラーメッセージを〈実行 `directory`〉`/log/error.log` に出力する。

以下の章で示していく開発行程のシナリオは、以下である。

1. AllegroCL の eli を使用し、emacs 上で動作するアプリケーションを開発する。ターゲットとする OS は、CentOS5 とする。
2. このアプリケーションを配布することになったため、AllegroCL が install されていないマシン上でも動作させる。
3. このアプリケーションを WindowsXP に移植する。
4. CentOS5 版アプリケーションを、デーモンプロセスとして動作させる。
5. このデーモンプロセス版を MacOSX に移植する。
6. 商用の処理系である AllegroCL ではなく、Free の処理系である CMUCL に移植する。デーモンプロセスである必要はない。ターゲットとする OS は、CentOS5 のみとする。

それぞれの行程順に、サンプルプログラムがどう変化していくかを見ていく。

4.1.1 emacs 上で動作するアプリケーションを開発

emacs 上での開発は、アプリケーション本体のメイン機能を実現することが目的である。

```
=== observe-directory.cl ============================================================
(defpackage "SAMPLE"
  (:use "COMMON-LISP")
  (:export "*OBSERVE-DIRECTORY*" "*LOG-DIRECTORY*" "*OBSERVE-INTERVAL*" "MAIN"))

(in-package "SAMPLE")

;;; 監視する directory の PATHNAME object
(defvar *observe-directory* (pathname "/usr/local/share/"))
;;; ログ directory
(defvar *log-directory* (pathname "./log/"))
;;; 監視間隔 (秒)
(defvar *observe-interval* 5)

;;; ログファイルの NAME
(defvar *history-log-name* "history.log")
(defvar *error-log-name* "error.log")

;;;
;;; ログ出力用の timestamp 文字生成関数
;;;
(defun timestamp ()
  (multiple-value-bind (s m h dd mm yy)
      (decode-universal-time (get-universal-time))
    (format nil "~4,'0D-~2,'0D-~2,'0D ~2,'0D:~2,'0D:~2,'0D" yy mm dd h m s)))
```

```
;;;
;;; 監視を行なうメイン loop
;;;
(defun observing-loop (directory interval log-file)
  (loop with prev-file-list = nil
      as current-file-list = (directory directory)
      as add-file-list = (set-difference current-file-list prev-file-list
                                         :test #'string-equal
                                         :key #'file-namestring)
      as del-file-list = (set-difference prev-file-list current-file-list
                                         :test #'string-equal
                                         :key #'file-namestring)
      when (or add-file-list del-file-list) do
        (with-open-file (stream log-file :direction :output
                         :if-does-not-exist :create :if-exists :append)
          (format stream "--- ~A ---~%" (timestamp))
          (format stream "~:[~;ADD:~%~{  ~A~%~}~]" add-file-list add-file-list)
          (format stream "~:[~;DEL:~%~{  ~A~%~}~]" del-file-list del-file-list))
      do (setq prev-file-list current-file-list)
         (sleep interval)))

(defun main ()
  (let ((error-log (merge-pathnames *error-log-name* *log-directory*))
        (history-log (merge-pathnames *history-log-name* *log-directory*)))
    (handler-bind ((error (lambda (condition)
                            (with-open-file (stream error-log
                                             :direction :output
                                             :if-does-not-exist :create
                                             :if-exists :append)
                              (format stream "--- ~A ---~%" (timestamp))
                              (format stream "ERROR: ~A~%" condition)
                              (force-output stream)))))
      (observing-loop *observe-directory* *observe-interval* history-log))))
================================================================================
```

```
=== AllegroCL on emacs の実行ログ ==============================================
CL-USER(2): (compile-file "observe-directory.cl" :load-after-compile t)
;;; Compiling file observe-directory.cl
;;; Writing fasl file observe-directory.fasl
;;; Fasl write complete
#P"/home/chika/sample/observe-directory/observe-directory.fasl"
NIL
NIL
CL-USER(3): (sample:main)
================================================================================
```

この状態で、/usr/local/share にファイルの追加と削除を何度か行なった。history.log に出力された内容は以下である。

```
=== 出力された history.log の内容 ==============================================
--- 2008-08-20 20:54:56 ---
ADD:
  /usr/local/share/uim
  /usr/local/share/anthy
  /usr/local/share/locale
  /usr/local/share/applications
  /usr/local/share/emacs
  /usr/local/share/doc
  /usr/local/share/info
  /usr/local/share/man
--- 2008-08-20 20:55:06 ---
ADD:
  /usr/local/share/dir
--- 2008-08-20 20:55:11 ---
DEL:
  /usr/local/share/dir
--- 2008-08-20 20:55:16 ---
ADD:
  /usr/local/share/foo
================================================================================
```

要求どおりの動作になることを確認した。

4.1.2 配布アプリケーションの作成

AllegroCL Enterprise 版では、配布可能なアプリケーションを作成することができる。この機能を用いる。

アプリケーション仕様として、以下の変更と追加をする。

1. (追加) コマンドの名前は sample とする。
2. (変更) ログの出力先は 〈install directory〉/log/ とする。
3. (追加) アプリケーションでエラーが発生した場合、その stacktrace である zoom 出力を標準出力に出力し[5]、監視は継続する。
4. (追加) アプリケーションは C-c により終了できる。

デーモン化にあたり、AllegroCL の起動スクリプト siteinit.cl を作成する。

```
=== siteinit.cl ======================================================================
(in-package "CL-USER")

;;; パスの変更
(setf sample:*log-directory* (pathname "sys:;log;"))

;;; stacktrace 出力関数 (<acl81>/src/autozoom.cl よりコピーして使用)
(defun zoom (stream &rest zoom-command-args
             &key (count t) (all t) &allow-other-keys)
  (with-standard-io-syntax
    (let ((*print-readably* nil)
          (*print-miser-width* 40)
          (*print-pretty* t)
          (top-level:*zoom-print-circle* t)
          (top-level:*zoom-print-level* nil)
          (top-level:*zoom-print-length* nil)
          (*terminal-io* stream)
          (*standard-output* stream))
      (apply #'top-level:do-command "zoom"
             :from-read-eval-print-loop nil
             :count count :all all zoom-command-args))))
```

[5]zoom を出力する理由は、一昨年のレポート
http://cl-www.msi.co.jp/solutions/knowledge/lisp-world/tutorial/condition-system.pdf
を参照のこと。

```
(defun main-loop ()
  (loop
    (ignore-errors
     (handler-bind ((excl:interrupt-signal (lambda (condition) (exit 0)))
                    (error (lambda (condition) (zoom *standard-output*))))
       (sample:main)))
    (sleep 10)))

(defun application-init-function ()
  (unless (probe-file sample:*log-directory*)
    (make-directory sample:*log-directory*))
  (main-loop))

(setq excl:*restart-actions*
  (append excl:*restart-actions*
          (list (lambda () (application-init-function)))))
================================================================================

=== AllegroCL on emacs の実行ログ (配布アプリケーションの生成) ====================
CL-USER(2): (excl:generate-executable "sample"
                                      '(:osi "observe-directory.fasl")
                                      :allow-existing-directory t
                                      :application-files '("siteinit.cl")
                                      :read-init-files '("siteinit.cl"))
; Autoloading for GENERATE-EXECUTABLE:
; Fast loading from bundle code/genexe.fasl.
;;; Compiling file /tmp/samplea63872119101.cl
;;; Writing fasl file /tmp/samplea63872119101.fasl
...
省略
...
CL-USER(3):
================================================================================
```

GENERATE-EXECUTABLE を実行した directory に作成された sample directory を tar で固める。これが配布物となる。これを別マシンに配布 (コピーして展開) して起動し、要求どおりの動作になることを確認した。

4.1.3 WindowsXP へ移植

WindowsXP 版では、監視する directory を c:\Program Files とする。

この移植に対する修正は、4.1.2 章の siteinit.cl のうち、監視 directory のみの変更である。OS 間で共通のアプリケーション仕様であるログを出力する directory については、CentOS5 版と WindowsXP 版の間で、完全に移植可能なプログラムであると言える。

```
=== siteinit.cl ============================================================
#+MSWINDOWS
(setf sample:*observe-directory* (pathname "c:/Program Files/"))
============================================================================
```

compile 方法と配布アプリケーションの作成方法は、4.1.1 章と 4.1.2 章の例と同様である。

以下が出力された history.log である。監視 directory に存在するファイルが多いため、出力を一部省略した ("... 省略" の部分)。

```
=== history.log ============================================================
=== 2008-08-13 21:12:05 ===
ADD:
  c:\Program Files\アタッシェケース
  c:\Program Files\Zero G Registry
  c:\Program Files\ytk
  c:\Program Files\Yahoo!
...
省略
  c:\Program Files\ActivePDF
  c:\Program Files\acl81
  c:\Program Files\Accessories
  c:\Program Files\aaa
=== 2008-08-13 21:12:25 ===
DEL:
  c:\Program Files\aaa
=== 2008-08-13 21:12:30 ===
ADD:
  c:\Program Files\foo
============================================================================
```

WindowsXP 上でも、要求どおりの動作になることを確認した。

4.1.4 CentOS5 版のデーモンプロセス化

4.1.2 章のアプリケーションをデーモンプロセス化する。標準出力 (stacktrace 出力用) は、〈install directory〉/log/zoom.log に出力する。

この移植に関する修正は、siteinit.cl に対する以下の箇所だけである (完全なソースは、4.1.7 章であげる)。

```
=== siteinit.cl 変更箇所 ============================================================
;;; zoom 用ログファイル定義の追加
#-MSWINDOWS
(defconstant *zoom-file* "sys:;log;zoom.log")

;;; daemon になるための detach 関数
#-MSWINDOWS
(defun make-daemon ()
  (if (> (excl.osi:fork) 0)
      (exit 0)
    (let ((p (open *zoom-file* :direction :output
                   :if-does-not-exist :create :if-exists :append)))
      (excl.osi:detach-from-terminal :output-stream p :error-output-stream p))))

;;; detach の呼び出し追加
(defun application-init-function ()
  (unless (probe-file sample:*log-directory*)
    (make-directory sample:*log-directory*))
  #-MSWINDOWS
  (make-daemon)
  (main-loop))
=============================================================================
```

参考までに、この例の中で使用している fork と detach-from-terminal は、Windows 版ではサポートされていない。

4.1.5 MacOSX へ移植

監視する directory は、CentOS5 版と同じとする。

この移植に対する修正は必要なく、4.1.1 章と 4.1.2 章の例と同様に、observe-directory.cl の compile と GENERATE-EXECUTABLE を実行するだけである。

MacOSX 上でも、要求どおりの動作になることを確認した。

4.1.6 CMUCL へ移植

この移植に関する修正は、ANSI Common Lisp ので記述した observe-directory.cl に関しては必要ない。siteinit.cl は AllegroCL 用の起動スクリプトであり使えないため、CMUCL における compile と起動のためのスクリプト cmu-init.cl を作成する。

```
=== cmu-init.cl ========================================================================
;;; コンパイルと load
(compile-file "/home/nara/chika/report/2008/observe-directory/observe-directory.cl")
(load "/home/nara/chika/report/2008/observe-directory/observe-directory")

;;; log directory の設定 (∵ SYS: は処理系に依存する予約語であるため。
(setf sample:*log-directory* (make-pathname :directory '(:absolute "home" "chika" "log")))

;;; プログラムの起動
(sample:main)
=========================================================================================

=== CMUCL の実行ログ ===================================================================
* (load "/home/nara/chika/report/2008/observe-directory/cmu-init.cl")
...
起動メッセージは省略。
=========================================================================================
```

AllegroCL と同様、要求どおりの動作なると思ったのだが、directory の追加や削除をしても、その履歴が history.log に出力されなかった。

原因は、observe-directory.cl 内 OBSERVING-LOOP 関数の中で、DIRECTORY 関数を使用してファイルの一覧 を取得しているが (戻り値は PATHNAME object の list) 、この PATHNAME にセットされた値が、AllegroCL と CMUCL とで異なるためである。差分計算の値取得に用いた関数 FILE-NAMESTRING の結果が異なっている。

```
=== AllegroCL の directory 関数呼び出し結果 ======================================
CL-USER(2): (directory "/usr/local/share/")
(#P"/usr/local/share/info" #P"/usr/local/share/locale"
 #P"/usr/local/share/man" #P"/usr/local/share/graphviz")
CL-USER(3): (file-namestring (car *))
"info"
CL-USER(4): (describe (car **))
#P"/usr/local/share/info" is a structure of type PATHNAME.  It has these slots:
 HOST               NIL
 DEVICE             :UNSPECIFIC
 DIRECTORY          (:ABSOLUTE "usr" "local" "share")
 NAME               "info"
 TYPE               NIL
 VERSION            :UNSPECIFIC
 NAMESTRING         "/usr/local/share/info"
 HASH               NIL
 DIR-NAMESTRING     NIL
 PLIST              NIL
CL-USER(5):

=== CMUCL の directory 関数呼び出し結果 =======================================
* (directory "/usr/local/share/")

(#P"/usr/local/share/graphviz/" #P"/usr/local/share/info/"
 #P"/usr/local/share/locale/" #P"/usr/local/share/man/")
* (file-namestring (car *))

NIL
* (describe (car **))

#P"/usr/local/share/graphviz/" is a structure of type PATHNAME.
HOST: #<LISP::UNIX-HOST>.
DEVICE: NIL.
DIRECTORY: (:ABSOLUTE "usr" "local" "share" "graphviz").
NAME: NIL.
TYPE: NIL.
VERSION: NIL.
*
================================================================================
```

DIRECTORY 関数の関数仕様は *ANSI20.2.1* 章に記載されている。この中で、処理系が独自にキーワード引数を追加することを許している。AllegroCL では :DIRECTORIES-ARE-FILES キーワード引数を追加しており、default では T である。このため、file 名と同様、directory 名が name 構成要素に設定さ

れた PATHNAME が返されたのである。:DIRECTORIES-ARE-FILES 引数を NIL に指定すると、CMUCL と同じ結果を得ることができる。

さて、どのように対処するかだが、DIRECTORY 関数では処理系依存のキーワード引数が許されているため、ここを変更して対処するのは得策ではない。FILE-NAMESTRING ではなく NAMESTRING を使用するよう、修正することにする。

この修正により、CMUCL でも、要求どおりの動作になることを確認した。

4.1.7 最終的なサンプルコード

この章のまとめとして、CentOS5, WindowsXP, MacOSX, CMUCL(CentOS5) で動作するソースコードをあげておく。

```
=== アプリケーション本体の機能 observe-directory.cl ==============================
(defpackage "SAMPLE"
  (:use "COMMON-LISP")
  (:export "*OBSERVE-DIRECTORY*" "*LOG-DIRECTORY*" "*OBSERVE-INTERVAL*" "MAIN"))

(in-package "SAMPLE")

;;; 監視する directory の PATHNAME object
(defvar *observe-directory* (pathname "/usr/local/share/"))
;;; ログ directory
(defvar *log-directory* (pathname "./log/"))
;;; 監視間隔 (秒)
(defvar *observe-interval* 5)

;;; ログファイルの NAME
(defvar *history-log-name* "history.log")
(defvar *error-log-name* "error.log")

;;;
;;; ログ出力用の timestamp 文字生成関数
;;;
(defun timestamp ()
  (multiple-value-bind (s m h dd mm yy)
      (decode-universal-time (get-universal-time))
    (format nil "~4,'0D-~2,'0D-~2,'0D ~2,'0D:~2,'0D:~2,'0D" yy mm dd h m s)))
```

```
;;;
;;; 監視を行なうメイン loop
;;;
(defun observing-loop (directory interval log-file)
  (loop with prev-file-list = nil
      as current-file-list = (directory directory)
      as add-file-list = (set-difference current-file-list prev-file-list
                                         :test #'string-equal
                                         :key #'namestring)
      as del-file-list = (set-difference prev-file-list current-file-list
                                         :test #'string-equal
                                         :key #'namestring)
      when (or add-file-list del-file-list) do
        (with-open-file (stream log-file :direction :output
                         :if-does-not-exist :create :if-exists :append)
          (format stream "--- ~A ---~%" (timestamp))
          (format stream "~:[~;ADD:~%~{  ~A~%~}~]" add-file-list add-file-list)
          (format stream "~:[~;DEL:~%~{  ~A~%~}~]" del-file-list del-file-list))
      do (setq prev-file-list current-file-list)
         (sleep interval)))

(defun main ()
  (let ((error-log (merge-pathnames *error-log-name* *log-directory*))
        (history-log (merge-pathnames *history-log-name* *log-directory*)))
    (handler-bind ((error (lambda (condition)
                            (with-open-file (stream error-log
                                             :direction :output
                                             :if-does-not-exist :create
                                             :if-exists :append)
                              (format stream "--- ~A ---~%" (timestamp))
                              (format stream "ERROR: ~A~%" condition)
                              (force-output stream)))))
      (observing-loop *observe-directory* *observe-interval* history-log))))
================================================================================

=== AllegroCL 起動用スクリプト siteinit.cl ============================================
(in-package "CL-USER")

;;; パスの変更
#+MSWINDOWS
(setf sample:*observe-directory* (pathname "c:/Program Files/"))
(setf sample:*log-directory* (pathname "sys:;log;"))
```

```
;;; zoom 用ログファイル定義の追加
#-MSWINDOWS
(defconstant *zoom-file* "sys:;log;zoom.log")

;;; stacktrace 出力関数 (<acl81>/src/autozoom.cl よりコピーして使用)
(defun zoom (stream &rest zoom-command-args
             &key (count t) (all t) &allow-other-keys)
  (with-standard-io-syntax
    (let ((*print-readably* nil)
          (*print-miser-width* 40)
          (*print-pretty* t)
          (top-level:*zoom-print-circle* t)
          (top-level:*zoom-print-level* nil)
          (top-level:*zoom-print-length* nil)
          (*terminal-io* stream)
          (*standard-output* stream))
      (apply #'top-level:do-command "zoom"
             :from-read-eval-print-loop nil
             :count count :all all zoom-command-args))))

(defun main-loop ()
  (loop
    (ignore-errors
     (handler-bind ((excl:interrupt-signal (lambda (condition) (exit 0)))
                    (error (lambda (condition) (zoom *standard-output*))))
       (sample:main)))
    (sleep 10)))

;;; daemon になるための detach 関数
#-MSWINDOWS
(defun make-daemon ()
  (if (> (excl.osi:fork) 0)
      (exit 0)
    (let ((p (open *zoom-file* :direction :output
                   :if-does-not-exist :create :if-exists :append)))
      (excl.osi:detach-from-terminal :output-stream p :error-output-stream p))))

(defun application-init-function ()
  (unless (probe-file sample:*log-directory*)
    (make-directory sample:*log-directory*))
  #-MSWINDOWS
  (make-daemon)
  (main-loop))
```

```
(setq excl:*restart-actions*
  (append excl:*restart-actions*
          (list (lambda () (application-init-function)))))
================================================================================

=== CMUCL 起動用 script cmu-init.cl ============================================
;;; コンパイルと load
(compile-file "/home/nara/chika/report/2008/observe-directory/observe-directory.cl")
(load "/home/nara/chika/report/2008/observe-directory/observe-directory")

;;; log directory の設定 (∵ SYS: は処理系に依存する予約語であるため。
(setf sample:*log-directory* (make-pathname :directory '(:absolute "home" "chika" "log")))

;;; プログラムの起動
(sample:main)
================================================================================
```

5　最後に

このレポートでは、プラットフォームに依存しないファイルの指定方法である PATHNAME を活用することで、移植性の高いアプリケーションを開発できることを示してきた。あくまでも ANSI Common Lisp 言語仕様で規定されているのは手段であり、プログラマが間違った使い方をすれば、移植性の低いアプリケーションになり得る。アプリケーションのメイン機能のプログラムの中で、ファイル名を直接参照するのと PATHNAME クラスの object を参照するのとでは大きな違いであることを理解し、この点に関しては労力を惜しまずにプログラミングすることが重要である。

また、更に移植性の高いアプリケーションを開発するためには、AllegroCL のように、より多くのプラットフォームをサポートしている処理系を開発環境として選ぶと良い[6]。ファイルの指定方法の違いの他、OS 間におけるシステムコールの違いについても処理系が吸収するためである。そして、そのソースファイルはプラットフォーム毎に管理するのではなく、全てのプラットフォーム上で動作するただ一組のソースファイルを管理すべきである。

多くのプラットフォームをサポートしている AllegroCL でバグと思える動作を見付けると、大抵、他のプラットフォーム版でも同じ動きをする。どの版も (コピーではない) 同一のソースで管理しているため、全ての版で同じ動作になるのだろう。ただ一組のソースファイルを管理するということは、1 箇所のバグを修正することで、全てのプラットフォーム版のバグ修正ができたことになる。こういう修正コストが最小限であることが、本当に優れた移植性の高いソフトウェアなのだと思う。

自分が開発しているアプリケーションも含め、数理システムが開発しているパッケージ類も、そういう優れたソフトウェアでありたい。

参考文献

[1] 『ANSI Common Lisp』
http://franz.com/support/documentation/8.1/ansicl/ansicl.htm

[2] Franz Inc., 『Pathname』
http://franz.com/support/documentation/8.1/doc/pathnames.htm

[3] 工藤千加子, 黒田寿男, 『Condition System の活用』
http://franz.com/support/documentation/8.1/doc/pathnames.htm

[6] AllegroCL は 18 個ものプラットフォームをサポートしている。http://jp.franz.com/base/map_platforms.html