F.G.H.
6/20/85

LispCourse #38: Files; Streams; Input/Output

Files

Files From the Programmer's Point of View

A file is just a data structure that is "outside of Lisp", e.g., on the local disk, on a file server, on the screen, etc.

> Because a file is not part of any Lisp virtual memory, it can (generally) be shared with other Lisp virtual memories or with other programming environments, etc.

Sinces files not part of Lisp, there are special protocols for creating and accessing files that are somewhat different from the way we access standard Lisp data structures.

What kind of data structure a file represents is largely determined by the protocols we use for accessing that file.

> At the lowest level, all files are just variable-length one-dimensional vectors of bytes (i,.e., 8-bit packets or integers between 0 & 255).
>
> However, with the proper choice of input/output statements in Lisp, a file can be made to look like any arbitrary data structure, e.g., a list of SExpressions, a TEdit text, a Sketch, etc.

When dealing with files in a Lisp program, you frequently shift between viewing the file as simply an unstructured vector of bytes and viewing the file as some higher-level data structure.

File Names

Every file has a file name. The file name basically tells Lisp where to find the file in the outside world.

The syntax and use of file names was discussed in LispCourse #15, pages 1 to 15.

The following are Interlisp functions that allow the programmer to construct and decompose file names:

(FILENAMEFIELD *FileName FieldName*) ÿ returns the part of *FileName* that is specified by *FieldName*. Allowable *FieldName*s are HOST, DIRECTORY, NAME, EXTENSION, and VERSION. (See LispCourse #15 for the semantics of these field names. HOST is the same as Device).

Example:

1_ (FILENAMEFIELD '{dsk}<halasz>lisp>init.lisp 'NAME)
init
2_ (FILENAMEFIELD '{dsk}<halasz>lisp>init.lisp 'DIRECTORY)
halasz>lisp

(UNPACKFILENAME *FileName*) ÿ returns *FileName* in prop list format, where the props are the allowable field names listed under FILENAMEFIELD.

Example:

3_ (UNPACKFILENAME '{dsk}<halasz>lisp>init.lisp;37)
(HOST dsk DIRECTORY halasz>lisp NAME init EXTENSION lisp VERSION 37)

(PACKFILENAME *FieldName1 FieldContents1 ... FieldNameN FieldContentsN*) ÿ returns a *FileName* constructed using the information in the *FieldName/FieldContents* pairs. The *FieldName*s should be chosen from among the field names listed under FILENAMEFIELD plus the atom BODY.

If a *FieldName* is specified twice, the first is used.

If any *FieldName* is BODY, then the corresponding *FieldContents* is unpacked (using UNPACKFILENAME) and the resulting *FieldName/FieldContents* pairs are used in place of the BODY/*FieldContents* pair.

Also, if *FieldName1* is a list, it is assumed to be a prop list of the format returned by UNPACKFILENAME, in which case PACKFILENAME is APPLYed to this list (ignoreing the remaining arguments!).

Example:

4_ (PACKFILENAME ’(HOST dsk DIRECTORY halasz>lisp NAME init EXTENSION lisp VERSION 37))

{dsk}<halasz>lisp>init.lisp;37

5_ (PACKFILENAME ’HOST ’{DSK} ’NAME ’FileX)

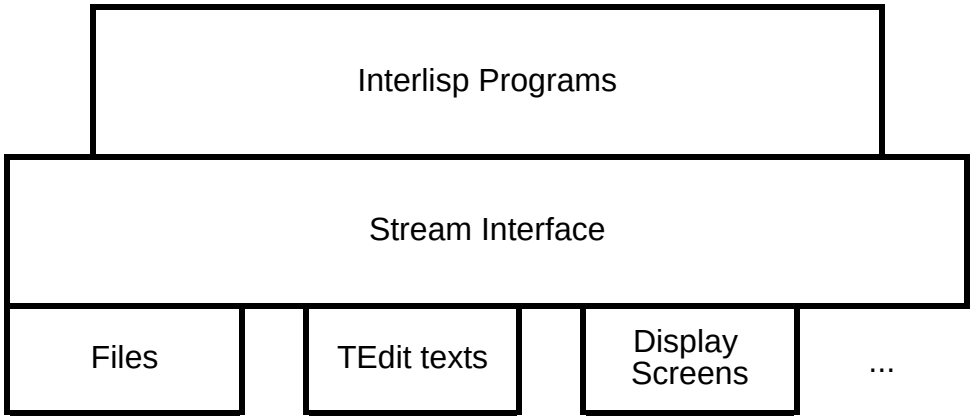
{DSK}FileX

Streams

In the old days, Interlisp dealt with files directly, i.e., anytime you needed to refer to a file you used its file name.

This convention has been replaced by the concept of ***streams*** in Interlisp-D.

Streams are a uniform interface between Interlisp and devices or data structures that exist in the "outside world" including files, TEdit texts, display screens, etc.



In this new scheme, Interlisp programs deal with files through the stream interface. For example, you generally refer to files using the stream that represents the file rather than by the name of the file.

A ***stream*** is a data type. Each stream instance represents an active interaction with an "external" device or data structure, e.g. a file.

Each instance stores with it all kinds of information about the status of the interaction and the status of the external device or data structure.

For example, a stream for a file stores information about the file's name, where you are in the file, how long the file is, etc.

Low-level Access to Files Using Streams

Opening and Closing

Before a file can be accessed, it must be ***opened***.

Opening a file, creates a stream for that file and caches in the stream instance all the necessary information. It also opens all the necessary communication pathways if the file is stored on a remote device such as a file server.

Opening a file also notifies the remote device to prevent access to the file by other users, if thats appropriate.

To open a file use:

(OPENSTREAM *FileName Access Recognition*) ž opens a file and creates a stream for it. OPENSTREAM returns the stream it creates. This stream should be used to refer to the file.

FileName is a standard Interlisp file name.

Access is one of INPUT, OUTPUT, APPEND, or BOTH.

If *Access* is INPUT, subsequent accesses to this file is limited to reading **from** the file

If *Access* is OUTPUT, subsequent access is limited to writing **to** the file. Moreover, if the file already exists it is erased.

If *Access* is APPEND, subsequent access is limited to writing **to** the file, but the current contents of the file (if any) are not erased.

If *Access* is BOTH, both reading and writing will be allowed.

Recognition is one of OLD, NEW, or OLD/NEW.

If *Recognition* is OLD, OPENSTREAM will look for an already existing file of the given file name.

If *Recognition* is NEW, OPENSTREAM will create a new version of the file with the given file name.

If *Recognition* is OLD/NEW, OPENSTREAM will look for an existing file first, but will create a new file if the old one doesn't exist.

Example:

```
6_ (SETQ FileX (OPENSTREAM '{DSK}Halasz.Lisp
'INPUT 'OLD)
{STREAM}#54,23123
```

(OPENP *FileNameOrStream Access*) ž returns the name of the file specified by *FileNameOrStream*, if that file is open with the access mode specified by *Access*. Returns NIL otherwise.

FileNameOrStream can be a file name or a stream or NIL. *Access* is one of the atoms described under OPENSTREAM or NIL.

If *Access* is NIL, then OPENP will return the file name if the file is open with any access mode.

If *FileNameOrStream* is NIL, then OPENP will just return a list of all open files.

Example:

```
7_ (SETQ FileX (OPENSTREAM '{DSK}Halasz.Lisp
'INPUT 'OLD)
{STREAM}#54,23123
8_ (OPENP Filex)
{DSK}Halasz.Lisp;23
9_ (OPENP '{DSK}Halasz.Lisp)
{DSK}Halasz.Lisp;23
10_ (OPENP Filex 'OUTPUT)
NIL
```

Once you are finished accessing a file, you should ***close*** the file.

Closing the file uncaches the information stored in the stream, deletes the stream, closes the communication pathways, and frees the file to be used by other users.

To close a file use:

(CLOSEF? *Stream*) ž closes the file specified by *Stream*, if that file is open. Returns the name of the file that it closed.

Stream should be a stream on an open file.

Example:

11_ (CLOSEF? Filex)
{DSK}Halasz.Lisp;23
12_ (CLOSEF? Filex)
NIL

(CLOSEALL) ž closes all currently open files. Returns a list of the names of the files that were closed.

File Pointers

Every open file has a ***file pointer*** associated with it.

The file pointer indicates the position in the file at which the next read or write will take place.

After each access to the file, the file pointer is updated to indicate the next position in the file (i.e., the next place to start reading or the next place to write).

Positions are measured in bytes, where the first byte in the file is position 0. So the file pointer varies between 0 and the one less than the length of the file.

When a file is first opened, the file pointer is set to 0 except if the access mode is append in which case the file pointer is set to the position of the end of the file.

At any point you can read the value of the file pointer to figure out where you are in the file.

For files that are on devices that support random access (e.g., local disk, IFS file servers, floppies but NOT NS file servers), you can set the file pointer to any arbitrary location in the file. The next read or write will then take place at this location.

The following functions read the file pointer:

(GETFILEPTR *Stream*) ÿ Returns the current file pointer for the open file represented by *Stream*, which should be a stream datatype.

(GETEOFPTR *Stream*) ÿ Returns the file pointer value for the location at the end of the open file represented by *Stream* (i.e., the number of bytes in the file).

The following functions can change the file pointer:

(RANDACCESSP Stream) ÿ Returns the file name of *Stream*, if that file is random accessable. NIL otherwise. *Stream* should represent an open file. If RANDACCESSP returns the file name, then SETFILEPTR can be used on the file.

(SETFILEPTR *Stream Position*) ž Sets the file pointer for the file represented by *Stream* to be *Position*. Position is any positive integer.

(SETFILEPTR *Stream* -1) is a special case meaning to set the file pointer to the end of the file.

SETFILEPTR results in s an error if the file is not RANDACCESSP.

(FILEPOS *Pattern Stream Start End Skip*) ž analogous to STRPOS (See LispCourse #28, page 18). Searches through the file referenced by *Stream* looking for any sequences of characters that matches the characters in string *Pattern*. If a match is found, FILEPOS sets the file pointer to the file position where the match starts and returns this position as its value. If no match is found, FILEPOS returns NIL and the file pointer is unchanged

If *Start* is specified, the search begins at file position *Start*, otherwise search starts at the current file pointer.

If *End* is specified, the search terminates at file position *End* (if no match has been found), otherwise search ends at the end of the file.

If *SkipChar* is specified, any instance of *SkipChar* in the *Pattern* string will match any character in the file. (*SkipChar* is the wildcard character).

Reading and Writing

The following two functions are used to read/write a byte from/to an open file:

(BIN *Stream*) ž Reads the next byte, i.e., the byte right after the file pointer and returns it. The file pointer is updated one byte. The byte returned is a number between 0 and 255.

(BOUT *Stream Byte*) ž Write *Byte* at the next position in the file, i.e., the position right after the file pointer. The file pointer is updated one byte. *Byte* should be a number between 0 and 255.

Example

The following is an implementation of a COPYBYTES function that copies the specified bytes from a source file to a destination file:

```
(DEFINEQ (COPYBYTES
    (LAMBDA (SourceFile DestFile Start End)
        (LET  (StopLoc
                    (SourceStream (OPENSTREAM SourceFile
                    'INPUT 'OLD))
                    (DestStream  (OPENSTREAM DestFile 'OUTPUT
                    'NEW)))
            (* * Set up correct start and stop pointers)
            (AND (NOT Start)(SETQ Start 0))
            (AND (NOT End)(SETQ End (GETEOFPTR
            SourceStream)))
```

```
(COND
    ((NOT (GREATERP Start End)) (ERROR "Error
    Msg")
(SETQ StopLoc (MIN (GETEOFPTR SourceStream)End))
(* * Move to start location)
(COND
    ((RANDACCESSP SourceStream)
        (SETFILEPTR SourceStream Start))
    (T (FOR Index FROM 0 TO (SUB1 Start)
        DO (BIN SourceStream))))
(* * Copy bytes until stop location)
(FOR Index FROM Start TO StopLoc
    DO (BOUT DestStream (BIN SourceStream)))
(* * Close files and return)
(LIST (CLOSEF? SourceStream) (CLOSEF?
DestStream))))))
```

Higher-level File Input and Output

For most applications, accessing files a byte at a time is unnecessarily detailed.

Interlisp provides a number of higer-level input/output routines that can read and write Lisp objects (i.e., SExpressions) rather than bytes.

These higher-level routines are, of course, simply functions that eventually call BIN and BOUT to access the file.

They just provide an more convenient interface to files for most applications.

Higher-level Input Functions

The standard input function is READ:

(READ *Stream*) ÿ reads and returns the next SExpression (litatom, number, list, string, etc.) from the file referenced by *Stream*. *Stream* must reference an open file.

READ will start at the current file position and skip over white space characters (i.e., space, tab, carriage return) until the first non-space character. It will the read in the SExpression that begins at this character.

The interpretation of SExpressions is done using the same rules as in the Lisp Exec ž lists are bounded by parentheses, strings are bounded by double quotes, atoms are bounded by white space or by the start/end of a list or string, numbers are atoms containing only digits, etc.

At the end of the READ, the file pointer for the file is set just after the last character of the SExpression.

(SKREAD *Stream*) ž moves the file pointer for the file referenced by *Stream* ahead as if a READ had been done, but does not actually read anything. SKREAD returns NIL.

Used for skipping over things you don't actually want to read.

Some more specialized input functions are:

(READC *Stream*) ž reads and returns the next character from the file referenced by *Stream*. The file pointer is moved ahead by 1.

READC reads all characters alike, ignoring the special effects of characters like double quotes and parentheses that would affect READ.

READC is like BIN, except it returns a character (i.e., a single character atom) rather than a byte (i.e., a number).

(RATOM *Stream*) ž reads and returns one atom from the file referenced by *Stream*. *Stream* must reference an open file. Works like READ, except that parentheses and double quotes are considered to be single character atoms.

Example: If the next SExpression on the file is (FOO), then RATOM would return the atom %(. But if the next SExpression were ABC, then RATOM would return the atom ABC.

(RSTRING *Stream*) ž reads characters from the file referenced by *Stream* up to but not including the next white space character, parentheses, or double quote. Returns the characters read as a string.. *Stream* must reference an open file. Sets the file pointer to right after the last character read.

Example: If the next character on the file on the file is a space, then RSTRING would return the null string "". But if the next characters were ABC, then RSTRING would return the string "ABC".

Finally, it is some times desireable to read a whole file in a single shot using the following function:

(READFILE *StreamOrFileName*) ž reads SExpressions from the file referenced by *StreamOrFileName* up to but not including the first occurrance of the atom STOP, or until the end of the file is reached if no STOP is encountered. Returns a list of the SExpressions read.

StreamOrFileName will be opened if necessary.

READFILE uses READ to do its reading.

Higher-level Output Functions

Interlisp has a number of functions for printing SExpressions onto files.

Note that these functions just print the print name of each lisp object on the file.

This is no problem for atoms, lists, numbers, and strings.

But, arrays and datatype have print names which are particularly non-informative. In order to print these kinds of objects, you generally have to decompose them and print their parts separately using your own functions.

When writing outSExpressions on a file, you have a to make a decision as to how the SExpressions should be written:

· in a way that can be read back into Lisp using READ

· in a way that is more human-readable but cannot be read back in by READ.

Example:

When you print a string you can print the double quotes or not.

If you print the double quotes, then READ can recognize the characters as a string when it is reading the file later.

If you don't print the double quotes, then your output may look nicer, but READ will not be able to recognize the charactres as a string when it tries to later read the file.

Interlisp provides higher-level output functions for both of these modes.

General Printing

The basic output functions are:

(PRIN1 *SExpression Stream*) ž prints SExpression on the open file referenced by *Stream*. Printing is done in a way that does not necessarily make it possible to READ the SExpression, e.g., the double quotes are omitted from strings and spaces inside of atoms are not escaped.

Examples:

The atom Foo% Bar is printed as Foo Bar

The string "ABC" is printed as ABC

The list (A B C) is printed as (A B C)

(PRIN2 *SExpression Stream*) ž prints SExpression on the open file referenced by *Stream*. Printing is done in a way that makes it possible to READ the SExpression, e.g., the double quotes are printed for strings and spaces inside of atoms are escaped.

Examples:

The atom Foo% Bar is printed as Foo% Bar

The string "ABC" is printed as "ABC"

The list (A B C) is printed as (A B C)

(TERPRI *Stream*) ž prints a carriage return on the open file referenced by *Stream*.

(SPACES *N Stream*) ž prints *N* spaces on the open file referenced by *Stream*.

(PRINT *SExpression Stream*) ž equivalent to (PRIN2 *SExpression Stream*) followed by a (TERPRI *Stream*).

Printing Numbers

The printing of numbers can be more precisely controlled using the following function:

(PRINTNUM *Format Number Stream*) ž prints *Number* on the open file referenced by *Stream* using the format specified by *Format*.

Format is a list structure with one of the following formats:

(FIX *Width Base Pad0Flg LeftFlushFlg*)

Indicates that *Number* is to be printed as a integer.

Width is the number of character spaces to reserve for the integer.

Base is the base in which the integer is to be printed. (Defaults to base 10.)

If *LeftFlushFlg* is NIL, then the number is right justified in the *Width* spaces. Otherwise, it is left justified and the unused spaces to the right are filled with blanks.

If *Pad0Flg* is T, then any part of the *Width* spaces that are to the left of the number are filled with zeros.

Examples: (Note: the | characters are for exposition only!):

(PRINTNUM ’(FIX 4) 100) prints | 100|.

(PRINTNUM ’(FIX 4 NIL T NIL) 100) prints |0100|.

(PRINTNUM '(FIX 4 NIL NIL T) 100) prints |100 |.

(FLOAT *Width DecWidth ExpWidth Pad0Flg*)

Indicates that *Number* is to be printed as a real number.

Width is the number of character spaces to reserve for the number.

DecWidth is the number of digits to appear to the right of the decimal point.

ExpWidth is non-NIL, specifies that the number should be printed in exponent format (i.e., scientific notation) with *ExpWidth* charactre spaces used for the exponent part.

PadChar if non-NIL specifies that the leading spaces to the left of the number are to be filled with zeros.

Examples: (Note: the | characters are for exposition only!):

(PRINTNUM '(FLOAT 4 2) 4.23) prints |4.23|.

(PRINTNUM '(FLOAT 5 2) 4.23) prints | 4.23|.

(PRINTNUM '(FLOAT 5 2 NIL 22) 4.23) prints |04.23|.

(PRINTNUM '(FLOAT 5 1 NIL 22) 4.23) prints |004.2|.

PRINTOUT

Finally, any reasonably complex output will involve lots of PRIN1s, TERPRIs, PRINTNUMs, etc. in very complex combinations. The PRINTOUT CLISP statement provides a simpler (sometimes) interface to all of this.

PRINTOUT is very complex and we will cover on a small bit of it here. See section 6.5 of the IRM for more details.

PRINTOUT has the format:

(PRINTOUT *Stream Command1 Command2* ...)

Stream references an open file.

Each *CommandI* is either one of the printing commands discussed below or an arbitrary Lisp SExpression.

PRINTOUT iterates through the *CommandIs*. If it encounters a command, it carries out that command. Otherwise, it prints the SExpression using PRIN1.

Note that numbers are PRINTOUT commands. Therefore the only way to print numbers is as the result of a coomand!

Some of the commands are:

.SP *N* ž print *N* spaces

T ž print a carriage return

.SKIP N ž skip N lines, i.e., print N returns

.PAGE ž print a form feed character

.FONT *FontSpec* ž change the font for printing to the file. *FontSpec* is a font list like (TIMESROMAN 12 BOLD).

.P2 *SExpr* ž PRIN2s SExpr

.FR ÿ***N SExpr*** ž PRIN1s *SExpr* right-flushed in the next *N* character positions.

.CENTER ÿ***N SExpr*** ž PRIN1s *SExpr* centered in the next *N* character positions.

.I*Format Number* žprints *Number* as an integer using *Format* as a format spec. *Format* is like the FIX format in

PRINTNUM, except that rather than parts of a list, you use atoms separated by periods.

Example: .I5.NIL.T is the same as (FIX 5 NIL T) in PRINTNUM.

.F*Format Number* žprints *Number* as an real number using *Format* as a format spec. *Format* is like the FLOAT format in PRINTNUM, except that rather than parts of a list, you use atoms separated by periods.

Example: .F5.2 is the same as (FLOAT 5 2) in PRINTNUM.

 # *SExpr* ž evaluates SExpr for side-effect only. Nothing is printed except as a result of the evaluation. USed to tailor your own PRINTOUT commands.

 # 'XXX will print nothing.

 # (PRIN1 'XXX Stream) will print XXX on the file.

Eamples:

(PRINTOUT Stream "This is Line 1" T "This is Line 2" T)

prints

This is Line 1

This is Line 2

(PRINTOUT Stream .I5 445 .SP 6 .F3.2 0.3456 T)

prints

| *445* *.34*|

Miscellaneous Considerations

The File T

There is one special input file in the system known as T. T is both an input and an output file. On the input end, it is the keyboard. On the output end, it is the Lisp Exec window.

You can use T as the Stream argument to any of the Lisp input/output functions discussed above.

Example:

(PRINT "Hello" T) prints Hello in the Exec window.

(READ T) reads the next SExpression the user types in.

When reading from T, the functions READ, RATOM,READC, etc. all wait for the user to type something before returning.

Line Buffering When Reading Keyboard Input

The file T and all other input that comes from the keyboard is subject to line buffering, i.e., the user type-in is held in a buffer until a carriage return is typed.

Until the return is typed, the user input is not available to be read by a program.

Once the return is typed, the user type-in can be read by a program.

Line buffering can be turned off using the function CONTROL:

(CONTROL T) ž turns the line-buffering off.

(CONTROL NIL) ž turns the line buffering on.

With line-buffering off, input can be read by the program immediately after it is typed in.

Example: If you want to do something immediately after the user types the next character, you should turn line buffering off and wait for the character press using READC or BIN as follows:

```
(DEFINEQ (WaitForChar
        (LAMBDA NIL
                (* * Turn off buffering)
                (CONTROL T)
                (PROG1
                        (READC T)(* * wait for a character press)
```

```
                                        (CONTROL NIL) (* * Turn on buffering
                                        again)))))
```

Default Input/Output File

Almost all of the Lisp input/output functions will accept NIL as their *Stream* argument. In this case they will use the current default input or output file.

Initially, the default input and output is set to T, i.e., the default input file is the keyboard and the default output file in the Lisp Exec window.

The following functions can be used to change the default files:

(INPUT *Stream*) ž makes the open file referenced by *Stream* be the default input file. Returns the old default input file. If *Stream* is NIL, just returns the default input file without changing it.

(OUTPUT *Stream*) ž makes the open file referenced by *Stream* be the default output file. Returns the old default output file. If *Stream* is NIL, just returns the default output file without changing it.

References

Input/Output is covered in Chapter 6 and Section 18.16 of the IRM.

Beware, however, this documentation is somewhat out of date. In particular, streams are not covered. Instead, files are said to be referenced by full file name, i.e., in the old manner.