File handling and pipes

- Lisp provides a number of file handling functions and macros, with the usual “open file, access, close file” approach
- We’ll look at a couple of basic examples, there are many other ways to attack file handling
- We’ll also look at handling pipes, allowing our programs to direct output to other programs/commands or pull input from other programs/commands

File info functions

- (probe-file fname) – return t iff the file exists
- (file-author fname) – return name of file author/owner
- (file-write-date fname) – look up modification date
- (file-namestring fname) – extract just the name (useful where fname includes the path)
- (directory-namestring fname) – extract just the path (omit the filename)
- (delete-file fname) – delete the file

with-open-file

- Function to deal with file processing
  - expects first parameter to be a pair:

    (streamname filename)
  - remaining parameters are the actions to be performed on the file
- It will open the file (read mode by default), after which you use the stream name to access
- file closes at end of with-open-file

The stream setup

- (streamname filename ...options...)
- Opens the file and associates the strem
- Can accept a variety of optional param (often pairs) eg:
  - :direction :output
  - :if-exists :overwrite
  - :if-does-not-exist :create

i/o with the open stream

- To read/return a line from an input stream (nil at eof)

```
(read-line streamname nil)
```

- To write to an open output stream

```
(format streamname “blah blah blah”)
```

- To get the size of a file

```
(file-length streamname)
```

- To get current position in file (bytes from start)

```
(file-position streamname)
```

Output example

- Open a file and write something to it

```
(with-open-file (mystream “somefile” :direction :output)
    (format mystream “blah blah blah”))
```

- Open a file and read a line from it, store in global variable

```
(defvar content nil)
(with-open-file (s “afile”)
   (setf content (read-line s nil)))
(format t “Content read was ~A~%” content)
```

Loop to read entire file (by lines)

```
(with-open-file (s somefilename)
   (do
       ; loop var nextline;init value;update statement
       ((nextline (read-line s nil) (read-line s nil)))

       ; stopping condition/action
       ((null nextline)  (format t “All done!~%”))

       ; loop body, just displays current line
       (format t “Just read: ~A~%” nextline)
   ))
```

Open and pipes

- We can use open to create pipes through which we can read data produced by other programs or write data that other programs can use as input
- Open returns the opened pipe, we close later with close
- Example: send our output to be read by another program

```
(defvar mypipe (open “| apath/aprog” :direction :output))
(format pipe “this is what I am sending through the pipe”)
(close pipe)
```

Example: reading from a pipe

- Run another program and read the first line of output it produces, store in a variable named result

```
(defvar p (open "| anotherprog))
(setf result (read-line p nil))
(close p)
```

Example: read all output from ls

```
(let ((mypipe (open “| ls”)))
   (do
   ; local vars value/init/update
   ((nextline (read-line mypipe nil) (read-line mypipe nil)))

   ; stopping condition/action
   ((null nextline) (close pipe))

   ; body (just printing what was read)
   (format t “just read: ~A~%” nextline)))
```

unwind-protect

- Sometimes there is an action we want to guarantee takes place at the end of a sequence, even if earlier parts fail (e.g. we always want to close the pipes we open)
- Unwind-protect takes a sequence of actions as its parameters, and if any fail it still continues with the rest

```
(unwind-protect
   (let ((p (open “| somecmd”)))
   ....use pipe, but stuff might fail...
   (close-pipe p)))
```