**9. Input/Output**

**1. Introduction**

You should be able to write and execute programs using interactive front end of Scheme using the knowledge provided by previous chapters.

In this chapter, I will explain about input and output. Using this feature, you can read and write data from/to files.

**2. Input from files**

**2.1.**open-input-file**,**read-char**, and**eof-object?

The function (**open-input-file** **filename**) is available to open a file. This function return a port for input. The function (**read-char** **port**) is to read a character from the **port**. As this function returns eof-object when it reaches the end of the file (EOF), you can check it by using **eof-object?**. The function (**close-input-port** **port**) is to close the input port. The [code 1] shows a function that returns file contents as string.

[code 1]

(define (read-file file-name)

(let ((p (open-input-file file-name)))

(let loop((ls1 '()) (c (read-char p)))

(if (eof-object? c)

(begin

(close-input-port p)

(list->string (reverse ls1)))

(loop (cons c ls1) (read-char p))))))

For instance, the result shown in [example 1] is obtained by applying the [code 1] to a file [hello.txt]. As the newline character is represented by '\n', it is not easy to read. Function **display** is available, however, for formatting ([example 2]).

[hello.txt]

Hello world!

Scheme is an elegant programming language.

[example 1]

(cd "C:\\doc")

(read-file "hello.txt")

;Value 14: "Hello world!\nScheme is an elegant programming language.\n"

[example 2]

(display (read-file "hello.txt"))

Hello world!

Scheme is an elegant programming language.

;Unspecified return value

**2.2. Syntaxes**call-with-input-file**and**with-input-from-file

You can open a file for input using the syntax **call-with-input-file** or **with-input-from-file**. These syntaxes are convenient because they handle errors.

(**call-with-input-file** **filename** **procedure**)

It opens a file named **filename** for input. The **procedure** is a function that takes input port as an argument. The file should be closed explicitly because it is not closed when the control is returned from the **procedure** if the input port is potentially used again. The [code 1] can be rewritten like [code 2] usingcall-with-input-file.

[code 2]

(define (read-file file-name)

(call-with-input-file file-name

(lambda (p)

(let loop((ls1 '()) (c (read-char p)))

(if (eof-object? c)

(begin

(close-input-port p)

(list->string (reverse ls1)))

(loop (cons c ls1) (read-char p)))))))

(**with-input-from-file** **filename** **procedure**)

It opens a file named **filename** as the standard input. The **procedure** is a function with no argument. The file is closed when the control is returned from the**procedure**. [code 3] shows the rewritten function of [code 1] using with-input-from-file.

[code 3]

(define (read-file file-name)

(with-input-from-file file-name

(lambda ()

(let loop((ls1 '()) (c (read-char)))

(if (eof-object? c)

(list->string (reverse ls1))

(loop (cons c ls1) (read-char)))))))

**2.3.**read

The function (**read** **port**) reads a S-expression from the **port**. It is convenient to read contents with parentheses like [paren.txt].

[paren.txt]

'(Hello world!

Scheme is an elegant programming language.)

'(Lisp is a programming language ready to evolve.)

[code 4]

(define (s-read file-name)

(with-input-from-file file-name

(lambda ()

(let loop ((ls1 '()) (s (read)))

(if (eof-object? s)

(reverse ls1)

(loop (cons s ls1) (read)))))))

The following shows the result of reading paren.txt by s-read.

(s-read "paren.txt")

⇒ ((quote (hello world! scheme is an elegant programming language.))

(quote (lisp is a programming language ready to evolve.)))

**Exercise 1**

Write the function **read-lines** that returns a list of strings which correspond to each line of file contents. The newline character is represented by#\Linefeed in Scheme. Following is the result of applying this function to the hello.txt.

(read-lines "hello.txt") ⇒ ("Hello world!" "Scheme is an elegant programming language.")

**3. Output to files**

**3.1. Making a Port for output**

Similar functions to those for input are available to make output ports.

(**open-output-file** **filename**)

It opens a file for output and returns a output port.

**(close-output-port port)**

It closes the port for output.

**(call-with-output-file filename procedure)**

It opens a file named **filename** for output and calls **procedure**. The function **procedure** takes the port as an argument.

**(with-output-to-file filename procedure)**

It opens a file named **filename** as the standard output and calls **procedure**. The **procedure** is a function with no argument. The file is closed when the control is returned from the **procedure**.

**3.1. Functions for output**

Following functions for output are available. These functions output to the standard output if the **port** is omitted.

(**write** **obj** **port**)

It outputs the **obj** to the **port**. Strings are enclosed in double quotes and characters are combined with the #\.

(**display** **obj** **port**)

It outputs the **obj** to the **port**. Strings are **not** enclosed in double quotes and characters are **not** combined with the #\.

(**newline** **port**)

It begins a new line.

(**write-char** **char** **port**)

It outputs the **char** to the **port**.

**Exercise 2**

Write a function to copy files (my-copy-file).

**Exercise 3**

Write the function (print-lines) that takes arbitrary number of strings as arguments and outputs them to the standard output. The output strings should be separated by newline.

**4.Summary**

This chapter is short because Scheme has a minimum IO facility.

I will explain about assignment on Scheme in the next chapter.

**Answers for Exercises**

**Answer 1**

First, write the function group-list that separates a list by a separator. Then divide the list of characters read from the input file by #\Linefeed and convert them into strings. (see ; \*)

(define (group-list ls sep)

(letrec ((iter (lambda (ls0 ls1)

(cond

((null? ls0) (list ls1))

((eqv? (car ls0) sep)

(cons ls1 (iter (cdr ls0) '())))

(else (iter (cdr ls0) (cons (car ls0) ls1)))))))

(map reverse (iter ls '()))))

(define (read-lines file-name)

(with-input-from-file file-name

(lambda ()

(let loop((ls1 '()) (c (read-char)))

(if (eof-object? c)

(map list->string (group-list (reverse ls1) #\Linefeed)) ; \*

(loop (cons c ls1) (read-char)))))))

example:

(group-list '(1 4 0 3 7 2 0 9 5 0 0 1 2 3) 0)

;Value 13: ((1 4) (3 7 2) (9 5) () (1 2 3))

(read-lines "hello.txt")

;Value 14: ("Hello world!" "Scheme is an elegant programming language." "")

**Answer 2**

(define (my-copy-file from to)

(let ((pfr (open-input-file from))

(pto (open-output-file to)))

(let loop((c (read-char pfr)))

(if (eof-object? c)

(begin

(close-input-port pfr)

(close-output-port pto))

(begin

(write-char c pto)

(loop (read-char pfr)))))))

**Answer 3**

(define (print-lines . lines)

(let loop((ls0 lines))

(if (pair? ls0)

(begin

(display (car ls0))

(newline)

(loop (cdr ls0))))))