**4. Defining Functions**

**1. Introduction**

In the previous chapters, I have explained:

1. How to install MIT-Scheme.
2. How Scheme interpreters evaluate S-expressions.
3. Basic list operations.

In this chapter, I will explain how to define your own functions. As Scheme is a functional programming language, you construct your programs by building up small functions. Thus, understanding how to build and combine functions is critical to master Scheme.

As it is quite inconvenient to define functions on the front end, you usually edit codes by an editor and load them to the interpreter.

**2. How to define simple functions and to load them**

You use **define** to bind a symbol to a value. You can define any kind of global parameters, such as numbers, characters, lists, etc. and functions by this operator.

Let's edit the code shown in [code 1] by any kinds of editor (including Notepad), and save it as 'hello.scm' under a directory like 'C:\doc\scheme\'. If possible, save under the default directory of MIT-Scheme, which you have defined in chapter 1.

[code 1] (hello.scm)

; Hello world as a variable

(define vhello "Hello world") ;1

; Hello world as a function

(define fhello (lambda () ;2

"Hello world"))

Next, give the following command to the Scheme interpreter.

(cd "C:\\doc\\scheme")

;Value 14: #[pathname 14 "c:\\doc\\scheme\\"]

(load "hello.scm")

;Loading "hello.scm" -- done

;Value: fhello

By this, hello.scm is loaded to the scheme interpreter. If the current directory of the interpreter has been set to the directory where the script is, you don't need the first line. Then give the following command to the interpreter.

vhello

;Value 15: "Hello world"

fhello

;Value 16: #[compound-procedure 16 fhello]

(fhello)

;Value 17: "Hello world"

The **define** is a operator to declare variables and takes two arguments. The operator declares to use the first argument as a global parameter and binds it to the second argument. Thus, the line at ;1 in [code 1] means that it declares a global parameter vhello and binds it to "Hello world".

On the other hand, the line at ;2 declares a procedure that returns "Hello world".  
The **lambda** is a special form to define procedures. The lambda takes more than one arguments and the first argument is the list of parameters that the procedure takes as arguments. As fhello takes no argument, the list of the parameters is empty.

Give vhello to the interpreter then it returns the value, "Hello world". if you give fhello, it also returns the value like,   
**#[compound-procedure 16 fhello]**,  
which means that the Scheme interpreter treats procedures and conventional data type in a same way. As explained in [the previous chapter](http://www.shido.info/lisp/scheme3_e.html#cons), the Scheme interpreter manipulates all the data by their addresses in the memory space and any kinds of object existing in the memory can be treated in the same way.   
To call fhello as a procedure, you should bracket off the symbol like (fhello). Then the interpreter evaluates it and returns "Hello world" taking [steps shown in chapter 2](http://www.shido.info/lisp/scheme2_e.html#eval).

**3. Defining functions with parameters**

Put a list of parameters after lambda to define functions with parameters. Save the code shown in [code 2] as farg.scm under the same directory where hello.scmis.

[code 2] (farg.scm)

; hello with name

(define hello

(lambda (name)

(string-append "Hello " name "!")))

; sum of three numbers

(define sum3

(lambda (a b c)

(+ a b c)))

After saving it, load the file into the interpreter and call the functions.

(load "farg.scm")

;Loading "farg.scm" -- done

;Value: sum3

(hello "Lucy")

;Value 20: "Hello Lucy!"

(sum3 10 20 30)

;Value: 60

**hello**

Function hello takes one argument (name), appends "Hello ", name, and "!" and returns it.  
A pre-defined function **string-append** takes arbitrary number of string arguments and returns a appended string.

**sum3**

It takes three arguments and returns the sum of them.

**4. A short form for function definitions**

Even, using lambda is the orthodox way of defining functions. you can use a short form shown in [code 3].

[code 3]

; hello with name

(define (hello name)

(string-append "Hello " name "!"))

; sum of three numbers

(define (sum3 a b c)

(+ a b c))

In this form, functions are defined in the form that they are called. [code 2] and [code 3] are identical. Some people don't like the short way of defining functions, but I use it in this tutorials, because this way makes code shorter.

**Exercise 1**

Write following functions. They are quite simple but useful.

1. A function that add 1 to the argument.
2. A function that subtract 1 from the argument.

**Exercise 2**

Let's write a function to calculate flying distances with following steps.

1. Write a function that convert the unit of angles from degree to radian.  
   180 degree is π radian. π is defined by  
   (define pi (\* 4 (atan 1.0)))
2. Write a function that calculates a distance of moving at a constant velocity (**vx**) in **t** seconds.
3. Write a function that calculates a duration till the object reach the ground that is launched with vertical velocity of **vy**.  
   Ignore air drag and use 9.8 m s-2 as the value of acceleration of gravity (*g*).   
   hint: As the vertical velocity when the object reaches the ground is **-vy**,  
   2 vy = g t   
   where t is the falling duration.
4. Write a function that calculates a flying distance of a ball thrown with an initial velocity **v** and an angle **theta** degree using functions defined in questions 1--3.  
   Hint: First, convert degree to radian (say **theta1**). The horizontal and vertical initial velocities are represented by   
   **v** cos(**theta1**) and **v** sin(**theta1**), respectively. The duration of falling can be calculated by the function of the question 3. As horizontal velocity doesn't change, the distance can be calculated using the function of the question 2.
5. Calculate the distance of a ball thrown with a initial velocity of 40 m s-1 and an angle of 30 degree. It corresponds to the distance of a long cast by a professional baseball player having a strong arm.

**5. About editors**

I will mention about editors which is convenient to edit Scheme codes.

**5.1. Emacs**

Windows version of Emacs21 can be downloaded from <http://ftp.gnu.org/gnu/emacs/windows/>. Download emacs-21.3-bin-i386.tar.gz and unpack it.

You will find an executable named runemacs.exe under directory bin. Double click it to start the editor. Even the key mapping is quite different from that of Windows standard, it is reasonably user friendly because it is equipped with a menu bar and mouse control. You can customize it by editing the configuration file named .emacs. A Scheme mode is available, which understands pre-defined words and indents code properly by pressing **Ctrl-i** or **TAB**. In addition, the editor shows corresponding open parenthesis automatically, when a close parenthesis are typed.

On Windows operation system, emacs cannot call MIT-Scheme interactively. You should save the source code and load it by hand. On Unix and Linux, on the other hand, emacs can call MIT-Scheme interactively and editing codes can be done interactively.

**5.2. Edwin**

Edwin is an editor equipped with MIT-Scheme. It is like emacs18. As there is no menu bar and no mouse control, it is not user friendly very much. Only few use this editor and little information is available on the Web. However, you can edit your codes interactively using this editor. I use this editor to edit Scheme codes on Windows.

**How to use the Edwin**

Double click the icon of Edwin to start the Edwin. When Edwin has started, a default buffer named \*scheme\* appears, which corresponds to the \*scratch\* buffer of emacs. You can use \*scheme\* as a front end of the interpreter. S-expressions are evaluated by pressing **(Ctrl-x, Ctrl-e)**.

**a. Open and close files and terminating the editor**

Press **(Ctrl-x, Ctrl-f)** to open a file. If you give a file name which does not exist, a new file is created. As the initial directory is set to 'C:\', you should change directory before open files.

Press **(Ctrl-x, Ctrl-s)** to save files and press **(Ctrl-x, Ctrl-w)** to save files with different names. To exit from editor press **(Ctrl-x, Ctrl-c)**.

**b. Indenting**

Press **Ctrl-i** or **TAB** to indent.

**c. Cut, copy and paste**

As you cannot use the mouse, copy(cut)/paste is inconvenient a little bit. Do like as follows:

1. First, go to the starting position to be selected region by using arrow keys. Then press **Ctrl-SPACE**.
2. Then go to the ending position and press **Ctrl-w** or **Alt-w** to cut or copy the selected region, respectively.
3. Finally, go to the position where you want paste and press **Ctrl-y**.

**d. Evaluating S-expressions**

* **Alt-z** is a key bound to evaluating the S-expression beginning with define. It also bind the symbol and value.
* **Alt-:** is a key to evaluate S-expressions at the mini buffer. This is often used to test functions evaluated by **Alt-z**.
* At \*scheme\* buffer, any S-expression can be evaluated by **(Ctrl-x, Ctrl-e)**.

See <http://www-swiss.ai.mit.edu/projects/scheme/documentation/user_8.html> and you will get more information about Edwin. The same document is attached with the MIT-Scheme you have downloaded.

**6. Summary**

In this chapter, I mentioned how to define functions. A special form **define** is used to define functions and other global parameters. Editing source codes by proper editor (like emacs) and loading them is much convenient than defining functions directly on the front end of the interpreter.

In the next chapter, I will mention about branching.

**Answers for exercises**

**Answer 1**

; 1

(define (1+ x)

(+ x 1))

; 2

(define (1- x)

(- x 1))

**Answer 2**

Write codes like as follows:

; definition of pi

(define pi (\* 4 (atan 1.0)))

; degree -> radian

(define (radian deg)

(\* deg (/ pi 180.0)))

; free fall time

(define (ff-time vy)

(/ (\* 2.0 vy) 9.8))

; horizontal distance

(define (dx vx t)

(\* vx t))

; distance

(define (distance v ang)

(dx

(\* v (cos (radian ang))) ; vx

(ff-time (\* v (sin (radian ang)))))) ; t

After loading to the interpreter, the distance will be calculated by;

(distance 40 30)

;Value: 141.39190265868385

The function returns a reasonable value 141.4 m, which is slightly larger because of ignoring air drag.