**7. Repetition**

**1. Introduction**

I will explain about repetition in this chapter. By using repetition, you will be able to write 'ordinary' programs. Even **do** expression is available, recursion is often used for repetition in Scheme.

**2. Recursion**

A recursive function is a function that calls itself in its definition. Even it may sound strange, it is a usual way for looping. If you have an analogy comparing functions to machines, recursion makes no sense. As the function is, however, a procedure, it makes sense to call itself. For instance, let's consider literature survey. You may need to read the literatures (*cited-1*) that are cited in the literature you are reading now. Again, you may need to read other literatures cited in the *cited-1*. Thus, literature survey is a recursive process and you can repeat surveying until satisfying a certain condition (say, you are tired). Thus, an analogy that compares functions in programming languages to some kind of human activities (say, literature survey) is useful to understand recursive functions.

Calculating factorial is often used to explain recursion.

[code 1] Function fact that calculates factorials.

(define (fact n)

(if (= n 1)

1

(\* n (fact (- n 1)))))

(fact 5) is calculated like as follows:

(fact 5)

⇒ 5 \* (fact 4)

⇒ 5 \* 4 \* (fact 3)

⇒ 5 \* 4 \* 3 \* (fact 2)

⇒ 5 \* 4 \* 3 \* 2 \* (fact 1)

⇒ 5 \* 4 \* 3 \* 2 \* 1

⇒ 5 \* 4 \* 3 \* 2

⇒ 5 \* 4 \* 6

⇒ 5 \* 24

⇒ 120

(fact 5) calls (fact 4), (fact 4) calls (fact 3), then finally (fact 1) is called. (fact 5), (fact 4) ,.., and (fact 1) are allocated at different memory spaces and(fact i) stays there until (fact (- i 1)) returns a value, which wastes the memory space and takes more calculation time because of the overhead of function call.

However, recursive functions can express repetition in a simple manner. Further as lists are defined recursively, lists and recursive functions fit together. For instance, a function that makes all list items twice is defined like as follows. The function should return an empty list if the argument is an empty list to terminate the calculation.

(define (list\*2 ls)

(if (null? ls)

'()

(cons (\* 2 (car ls))

(list\*2 (cdr ls)))))

**Exercise 1**

Write following functions using recursion.

1. A function that counts the number of list items, my-length. (Function length is pre-defined.)
2. A function that summarizes numbers in a list.
3. A function that takes a list (**ls**) and an object (**x**) as arguments and returns a list removing **x** from **ls**.
4. A function that takes a list (**ls**) and and an object (**x**) and returns the first position of **x** in **ls**. The position is counted from 0. If **x** is not found in **ls**, the function returns #f.

**3. Tail Recursive**

Ordinary recursive function is not efficient because of wasting memory and function call overhead. On the contrary, tail recursive functions include the result as argument and returns it directory when the calculation finishes. Especially, as Scheme specification requires conversion of a tail recursive to a loop, there is no function call overhead.

[code 2] shows a tail recursive version of function fact shown in [code 1].

[code 2] fact-tail, tail recursive version of fact

(define (fact-tail n)

(fact-rec n n))

(define (fact-rec n p)

(if (= n 1)

p

(let ((m (- n 1)))

(fact-rec m (\* p m)))))

fact-tail calculates factorial like as follows:

(fact-tail 5)

⇒ (fact-rec 5 5)

⇒ (fact-rec 4 20)

⇒ (fact-rec 3 60)

⇒ (fact-rec 2 120)

⇒ (fact-rec 1 120)

⇒ 120

As fact-rec does not wait the result of other functions, it disappears from the memory space when it finishes. The calculation proceeds by changing argument of fact-rec, which is basically the same as loop. As mentioned previously, as Scheme convert a tail recursive to a loop, Scheme can do repetition without syntax for looping.

**Exercise 2**

Write following functions using tail recursive.

1. my-reverse that reverse the order of list items. (Function reverse is pre-defined.)
2. Summarizing items of a list consisting of numbers.
3. Converting a string that represents a positive integer to the corresponding integer, i.e. "1232" → 1232. Input error check is not required.  
   Hint:
   1. Character to number conversion is done by subtracting 48 from the ASCII codes of characters #\0 ... #\9. Function char->integer is available to get the ASCII code of a character.
   2. Function string->list is available to convert string to a list of characters.

**4. Named let**

The **named**let is available to express loop. [code 3] shows a function fact-let that calculates factorials using named let. The fact-let uses a named let expression (**loop**), instead of fact-rec shown in [code 2]. First it initializes parameters (**n1**, **p**) with **n** at the line marked with ; 1. These parameters are updated at the line marked with ; 2 after each cycle: Subtracting **n1** by one and multiplying **p** by (n1-1)

A named let is a conventional way to express loops in Scheme.

[code 3]

(define (fact-let n)

(let loop((n1 n) (p n)) ; 1

(if (= n1 1)

p

(let ((m (- n1 1)))

(loop m (\* p m)))))) ; 2

**Exercise 3**

Write followings by using named let.

1. Questions 3 and 4 of [exercise 1](http://www.shido.info/lisp/scheme7_e.html" \l "p1).
2. The function of [exercise 2](http://www.shido.info/lisp/scheme7_e.html#p2).
3. The function range that returns a list of numbers from 0 to **n** (not including **n**).

**5. letrec**

While it is similar to the named let, a name defined by letrec can refer itself in its definition. The letrec syntax is used to define complicated recursive functions. [code 4] shows a letrec version of fact.

[code 4]

(define (fact-letrec n)

(letrec ((iter (lambda (n1 p)

(if (= n1 1)

p

(let ((m (- n1 1)))

(iter m (\* p m))))))) ; \*

(iter n n)))

As shown at the line of ; \*, the local variable **iter** can refer itself in the definition of **iter**. Syntax letrec is a conventional way to define local functions.

**Exercise 4**

Write letrec version of [exercise 2](http://www.shido.info/lisp/scheme7_e.html#p2).

**5. do expression**

Syntax **do** is available for repetition, even it is not used frequently. The format is like as follows:

(**do** **binds** (**predicate** **value**)

**body**)

Variables are bound in the **binds** part, if the **predicate** is true, it escapes from the loop and returns the **value**, otherwise the looping continues.

The format of the **binds** part is like as follows:

[**binds**] → ((p1 i1 u1) (p2 i2 u2) ... )

Variables p1, p2 ... are initialized with i1, i2 ... and updated with u1, u2 ... after each cycle.

[code 5] shows do version of fact.

[code 5]

(define (fact-do n)

(do ((n1 n (- n1 1)) (p n (\* p (- n1 1)))) ((= n1 1) p)))

The variables **n1** and **p** are initialized with **n** and **n** and subtracted by one and multiplying by (n1 - 1) after each cycle, respectively. The function returns **p**when **n1** becomes one.

I feel do is rather complicated than named let.

**Exercise 5**

Write a do version of [exercise 2](http://www.shido.info/lisp/scheme7_e.html#p2).

**6. Summary**

Now you can write ordinary programs using technique I have explained.

Usually **named let** is used for simple loops. On the other hand, **letrec** is used for complicated recursive local functions.

I will explain higher order functions in the next chapter. Higher order functions makes your code more Scheme like.

**Answers for Exercises**

**Answer 1**

; 1

(define (my-length ls)

(if (null? ls)

0

(+ 1 (my-length (cdr ls)))))

; 2

(define (my-sum ls)

(if (null? ls)

0

(+ (car ls) (my-sum (cdr ls)))))

; 3

(define (remove x ls)

(if (null? ls)

'()

(let ((h (car ls)))

((if (eqv? x h)

(lambda (y) y)

(lambda (y) (cons h y)))

(remove x (cdr ls))))))

; 4

(define (position x ls)

(position-aux x ls 0))

(define (position-aux x ls i)

(cond

((null? ls) #f)

((eqv? x (car ls)) i)

(else (position-aux x (cdr ls) (1+ i)))))

**Answer 2**

; 1

(define (my-reverse ls)

(my-reverse-rec ls ()))

(define (my-reverse-rec ls0 ls1)

(if (null? ls0)

ls1

(my-reverse-rec (cdr ls0) (cons (car ls0) ls1))))

;-------------------

; 2

(define (my-sum-tail ls)

(my-sum-rec ls 0))

(define (my-sum-rec ls n)

(if (null? ls)

n

(my-sum-rec (cdr ls) (+ n (car ls)))))

;--------------------

; 3

(define (my-string->integer str)

(char2int (string->list str) 0))

(define (char2int ls n)

(if (null? ls)

n

(char2int (cdr ls)

(+ (- (char->integer (car ls)) 48)

(\* n 10)))))

**Answer 3**

; 1

(define (remove x ls)

(let loop((ls0 ls) (ls1 ()))

(if (null? ls0)

(reverse ls1)

(loop

(cdr ls0)

(if (eqv? x (car ls0))

ls1

(cons (car ls0) ls1))))))

; 2

(define (position x ls)

(let loop((ls0 ls) (i 0))

(cond

((null? ls0) #f)

((eqv? x (car ls0)) i)

(else (loop (cdr ls0) (1+ i))))))

; 3

(define (my-reverse-let ls)

(let loop((ls0 ls) (ls1 ()))

(if (null? ls0)

ls1

(loop (cdr ls0) (cons (car ls0) ls1)))))

; 4

(define (my-sum-let ls)

(let loop((ls0 ls) (n 0))

(if (null? ls0)

n

(loop (cdr ls0) (+ (car ls0) n)))))

; 5

(define (my-string->integer-let str)

(let loop((ls0 (string->list str)) (n 0))

(if (null? ls0)

n

(loop (cdr ls0)

(+ (- (char->integer (car ls0)) 48)

(\* n 10))))))

; 6

(define (range n)

(let loop((i 0) (ls1 ()))

(if (= i n)

(reverse ls1)

(loop (1+ i) (cons i ls1)))))

**Answer 4**

; 1

(define (my-reverse-letrec ls)

(letrec ((iter (lambda (ls0 ls1)

(if (null? ls0)

ls1

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

(iter ls ())))

; 2

(define (my-sum-letrec ls)

(letrec ((iter (lambda (ls0 n)

(if (null? ls0)

n

(iter (cdr ls0) (+ (car ls0) n))))))

(iter ls 0)))

; 3

(define (my-string->integer-letrec str)

(letrec ((iter (lambda (ls0 n)

(if (null? ls0)

n

(iter (cdr ls0)

(+ (- (char->integer (car ls0)) 48)

(\* n 10)))))))

(iter (string->list str) 0)))

**Answer 5**

; 1

(define (my-reverse-do ls)

(do ((ls0 ls (cdr ls0)) (ls1 () (cons (car ls0) ls1)))

((null? ls0) ls1)))

; 2

(define (my-sum-do ls)

(do ((ls0 ls (cdr ls0)) (n 0 (+ n (car ls0))))

((null? ls0) n)))

; 3

(define (my-string->integer-do str)

(do ((ls0 (string->list str) (cdr ls0))

(n 0 (+ (- (char->integer (car ls0)) 48)

(\* n 10))))

((null? ls0) n)))