**5. Branching**

**1. Introduction**

In the previous chapter, I have explained how to define functions. In this chapter I will explain how to branch the procedures by conditions. This is an important step to write real programs.

**2. The if expression**

The **if** expression is to branch procedure into two parts. The format is like as follows:

(**if** **predicate** **then\_value** **else\_value**)

If the **predicate** is **true**, then **then\_value** otherwise **else\_value** is evaluated and the value goes out from the outermost parentheses of the if expression. **True**is any value except **false** which is represented by **#f**. The representing value of true is **#t**.

In R5RS, the false (#f) and the empty list ('()) are different object. However, in MIT-Scheme they are the same. This difference may be due to a historical reason; in R4RS, the previous standard, #f and '() are defined as the same object.

Thus, you should not use lists directory as predicates for the compatibility. Use function **null?** to see if the list is empty.

(null? '())

;Value: #t

(null? '(a b c))

;Value: () ;#f

Function **not** is available to negate predicates. This function takes exactly one argument and returns #t if the argument is #f otherwise #t.

The **if** expression is a special form because it does not evaluate all arguments. If the **predicate** is true, only **then\_value** is evaluated. On the other hand, if the **predicate** is false, only **else\_value** is evaluated.

Example: sum of geometric progression, whose first term, geometric ratio, and number of terms are **a0**, **r**, and **n**.

(define (sum-gp a0 r n)

(\* a0

(if (= r 1)

n

(/ (- 1 (expt r n)) (- 1 r))))) ; !!

In general, the sum of geometric progression is calculated by:

a0 \* (1 - rn) / (1 - r) (r ≠ 1)

a0 \* n (r = 1)

If the if expression evaluate all the arguments, the line commented with ; !! is evaluated even when r=1 and a devision by zero error occurs.

You can omit the **else\_value** term. In this case the returned value when the **predicate** is false is not specified. If you need #f when the **predicate** is false, you should write it explicitly.

The **then\_value** and **else\_value** should be one S-expression. If you need side effects, you should use **begin** expression. I will mention about begin in a later chapter.

**Exercise 1**

Make following functions. See [Section 5](http://www.shido.info/lisp/scheme5_e.html#predicate) to make predicates.

1. A function to return the absolute value of a real number.
2. A function to return the reciprocal of a real number. Make it return #f if the argument is 0.
3. A function to convert a integer to a graphical character of ASCII characters. The integers that can be converted to graphical characters are 33 – 126. Use**integer->char** to convert a integer to a character. Make it return #f if the given integer can not be converted to a graphical character.

**3. and and or**

The **and** and **or** are special forms to be used to combine conditions. Scheme's and and or are somehow different from those of conventional languages such as C. They do not return boolean (#t or #f) but returns one of given arguments. The **and** and **or** can makes your code shorter.

**3.1. and**

The **and** takes arbitrary number of arguments and evaluates them from left to right. It returns #f if one of the arguments is #f and the rest of arguments are not evaluated. On the other hand, if all arguments are not #f, it returns the value of the last argument.

(and #f 0)

;Value: ()

(and 1 2 3)

;Value: 3

(and 1 2 3 #f)

;Value: ()

**3.2. or**

The **or** takes arbitrary number of arguments and evaluates them from left to right. It returns the value of the first argument which is not #f and the rest of arguments are not evaluated. It returns the value of the last argument if it is evaluated.

(or #f 0)

;Value: 0

(or 1 2 3)

;Value: 1

(or #f 1 2 3)

;Value: 1

(or #f #f #f)

;Value: ()

**Exercise 2**

Make following functions:

1. A function that takes three real numbers as arguments. It returns the product of these three numbers if all them is positive.
2. A function that takes three real numbers as arguments. It returns the product of these three numbers if at least one of them is negative.

**4. cond expression**

Even all branchings can be expressed by if expression, you should nest it if the branching has a wide variety, which makes code complicated. The **cond**expression is available for such cases. The format of the cond expression is like as follows:

(**cond**

(**predicate\_1** **clauses\_1**)

(**predicate\_2** **clauses\_2**)

......

(**predicate\_n** **clauses\_n**)

(**else** **clauses\_else**))

In the cond expression, **predicates\_*i*** are evaluated from top to bottom, and **clauses\_*i*** is evaluated and returns from the cond expression if the **predicates\_*i***is true. Other clauses and predicates after *i* are not evaluated. If all of **predicates\_*i*** are false it returns the value of **clauses\_else**. You can write more than one S-expression in one clauses and the value of the last S-expression is the value of the clauses.

Example: Fee of a city-run swimming pool  
The fee of a city-run swimming pool of Foo city depends on the age of users (**age**):

* free if **age** ≤ 3 or **age** ≥ 65.
* 0.5 dollars for 4 ≤ **age** ≤ 6.
* 1.0 dollars for 7 ≤ **age** ≤ 12.
* 1.5 dollars for 13 ≤ **age** ≤ 15.
* 1.8 dollars for 16 ≤ **age** ≤ 18.
* 2.0 dollars for others.

The function that returns the fee of the city-run swimming pool is as follows:

(define (fee age)

(cond

((or (<= age 3) (>= age 65)) 0)

((<= 4 age 6) 0.5)

((<= 7 age 12) 1.0)

((<= 13 age 15) 1.5)

((<= 16 age 18) 1.8)

(else 2.0)))

**Exercise 3**

Make following functions.

1. The grade (A – D) of exam is determined by the score (**score**). Write a function that takes a score as an argument and returns a grade.
   1. A if **score** ≥ 80
   2. B if 60 ≤ **score** ≤ 79
   3. C if 40 ≤ **score** ≤ 59
   4. D if **score** < 40

**5. Functions that makes predicates**

I introduce some functions that make predicates. Such functions have names that end with '?'.

**5.1. eq?, eqv?, and equal?**

The **eq?**, **eqv?**, and **equal?** take exactly two arguments and are basic functions to check if the arguments are 'same'. These three functions are slightly different from each other.

**eq?**

It compares addresses of two objects and returns #t if they are same. In the following example, the function returns #t as str has the same address as itself. On the contrary, as "hello" and "hello" are stored in the different address, the function returns #f. Don't use eq? to compare numbers because it is not specified in R5RS even it works in MIT-Scheme. Use **eqv?** or **=** instead.

(define str "hello")

;Value: str

(eq? str str)

;Value: #t

(eq? "hello" "hello")

;Value: () ← It should be #f in R5RS

;;; comparing numbers depends on implementations

(eq? 1 1)

;Value: #t

(eq? 1.0 1.0)

;Value: ()

**eqv?**

It compares types and values of two object stored in the memory space. If both types and values are same, it returns #t. Comparing procedure (lambda expression) depends on implementations. This function cannot be used for sequences such as lists or string, because the value stored in the first address are different even the sequences looks same.

(eqv? 1.0 1.0)

;Value: #t

(eqv? 1 1.0)

;Value: ()

;;; don't use it to compare sequences

(eqv? (list 1 2 3) (list 1 2 3))

;Value: ()

(eqv? "hello" "hello")

;Value: ()

;;; the following depends on implementations

(eqv? (lambda(x) x) (lambda (x) x))

;Value: ()

**equal?**

It is used to compare sequences such as list or string.

(equal? (list 1 2 3) (list 1 2 3))

;Value: #t

(equal? "hello" "hello")

;Value: #t

**5.2. Functions that check data type**

Followings are type checking functions. All of them take exactly one argument.

**pair?**

It returns #t if the object consists of cons cells (or a cons cell).

**list?**

It returns #t if the object is a list. Be careful in that '() is a list but not a pair.

**null?**

It returns #t if the object is '().

**symbol?**

It returns #t if the object is a symbol.

**char?**

It returns #t if the object is a character.

**string?**

It returns #t if the object is a string.

**number?**

It returns #t if the object is a number.

**complex?**

It returns #t if the object is a complex number.

**real?**

It returns #t if the object is a real number

**rational?**

It returns #t if the object is a rational number.

**integer?**

It returns #t if the object is an integral

**exact?**

It returns #t if the object is **not** a floating point number.

**inexact?**

It returns #t if the object is a floating point number.

**5.3. Functions that compare numbers**

**=, <, >, <=, >=**

These functions takes arbitrary number of arguments. If arguments are ordered properly indicated by the function name, the functions return #t.

(= 1 1 1.0)

;Value: #t

(< 1 2 3)

;Value: #t

(< 1)

;Value: #t

(<)

;Value: #t

(= 2 2 2)

;Value: #t

(< 2 3 3.1)

;Value: #t

(> 4 1 -0.2)

;Value: #t

(<= 1 1 1.1)

;Value: #t

(>= 2 1 1.0)

;Value: #t

(< 3 4 3.9)

;Value: ()

**odd?, even?, positive?, negative?, zero?**

These functions take exactly one argument and return #t if the argument satisfies the property indicated by the function name.

**5.4. Functions that compare characters**

Functions **char=?**, **char<?**, **char>?**, **char<=?**, and **char>=?** are available to compare characters. See [R](http://www.schemers.org/Documents/Standards/R5RS/HTML/)[5](http://www.schemers.org/Documents/Standards/R5RS/HTML/)[RS](http://www.schemers.org/Documents/Standards/R5RS/HTML/) for detailed information.

**5.5. Functions that compare strings**

Functions **string=?** and **string-ci=?** etc are available. See [R5RS](http://www.schemers.org/Documents/Standards/R5RS/HTML/) for detailed information.

**6. Summary**

In this chapter, I have explained on branching. The **if** expression or **cond** expression are available for branching.

I will explain about the local variables in the next chapter.

**Answers for Exercises**

**Answer 1**

; 1

(define (my-abs n)

(\* n

(if (positive? n) 1 -1)))

; 2

(define (inv n)

(if (not (zero? n))

(/ n)))

; 3

(define (i2a n)

(if (<= 33 n 126)

(integer->char n)))

**Answer 2**

; 1

(define (pro3and a b c)

(and (positive? a)

(positive? b)

(positive? c)

(\* a b c)))

; 2

(define (pro3or a b c)

(if (or (negative? a)

(negative? b)

(negative? c))

(\* a b c)))

**Answer 3**

(define (score n)

(cond

((>= n 80) 'A)

((<= 60 n 79) 'B)

((<= 40 n 59) 'C)

(else 'D)))