Scheme Programming Basics/Elements

**Additional reference : https://www.scheme.com/tspl4/**

**Functions**

**1 (define fun**

**(lambda (arg1 arg2)**

**...))**

**2 (define (fun arg1 arg2)**

**...)**

**3 (fun value1 value2)**

**4 (apply fun (list value1 value2))**

**Functions are first-class objects in Scheme. They can be arguments to other functions and be returned by them. They can be assigned to variables. For example a function with two arguments *arg1* and *arg2* is defined in line 1 and line 2 is an abbreviation of it. Line 3 shows how functions are applied. Note that the function being applied is in the first position of the list while the rest of the list contains the arguments. The apply function will take the first argument and apply it to a given list of arguments, so the previous function call can also be written as seen on line 4.**

**In Scheme, functions are divided into two basic categories: procedures and primitives. All primitives are procedures, but not all procedures are primitives. Primitives are pre-defined functions in the Scheme language. These include +, -, \*, /, set!, car, cdr, and other basic procedures. Procedures are user-defined functions. In several variations of Scheme, a user can redefine a primitive. For example, the code**

**(define (+ x y)**

**(- x y))**

**or simply**

**(define + -)**

**actually redefines the + primitive to perform subtraction, rather than addition.**

***Another function example –***

***(define recpcl***

***(lambda (n)***

***(if (= n 0)***

***"Invalid denominator "***

***(/ 1 n))))***

**Here, recpcl is the short for reciprocal and it calculates 1/n where n is not equal to Zero for any number. If n is zero, it will return an error as “Invalid denominator”. And you can try this by typing:**

***(recpcl 3) => 1/3*  
*(recpcl 9/4) => 4/9*  
*(recpcl 0) =>"Invalid Denominator"***

**Lists**

Scheme uses the linked list data structure in the same form as it exists in Lisp. "list" builds a new linked list structure, for example:

(list 1 2 3) (list (list 1 2) 3)

" car" (pronounced: [ka`r] ) gives the value of the head node of the list, for example:

(car (list 1 2 3))

gives

1

and

(car (list (list 1 2) 3))

gives

(1 2)

" cdr" (pronounced "could-er" or ['kudər]) gives the list after the head node, for example:

(cdr (list 1 2 3))

gives

(2 3)

and

(cdr (list (list 1 2) 3)

gives

(3)

" cons" constructs a new list with a given car value and cdr list, for example:

(cons 1 (list 2 3))

gives

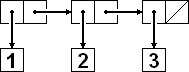
(1 2 3)

and

(cons (list 1 2) (list 3))

gives

((1 2) 3)

[](https://www.cs.mcgill.ca/~rwest/wikispeedia/wpcd/images/77/7779.png.htm)

Each node in the linked list is a cons cell, also called a pair. As the name pair implies, a cons cell consists of two values: the first one is the car, and the second is the cdr. For

(list 1 2 3)

there are three cons cells, or pairs. The first cons cell has the number 1 in the first slot, and a pointer to the second cons cell in the second. The second cons cell has the number 2 in the first slot, and a pointer to the third cons cell in the second slot. The third cons cell has the number 3 in the first slot and a null constant in the second slot. The null constant is usually represented by '() or (quote ()). The cons function constructs these cons cells, which is why

(cons 1 (list 2 3))

gives the list

(1 2 3)

If both of the arguments are not lists, then a pair is created, represented with a dot. For example

(cons 1 2)

gives

(1 . 2)

where the cons cell consists of 1 and 2 in its slots instead of a pointer to another cons cell in its second slot.

The names of the two primitive operations for decomposing lists, car and cdr, originally come from assembly language macros for the IBM 704; they stood for "contents of address register" and "contents of decrement register" respectively.

**Data types**

Other common data types in Scheme besides functions and lists are: integer, rational, real, complex numbers, symbols, strings, and ports. Most Scheme implementations also offer association lists, hash tables, vectors, arrays and structures. Since the IEEE Scheme standard and the R4RS Scheme standard, Scheme has asserted that all of the above types are *disjoint*, that is no value can belong to more than one of these types; however some ancient implementations of Scheme predate these standards such that #f and '() refer to the same value, as is the case in Common Lisp.

Most Scheme implementations offer a full numerical tower as well as exact and inexact arithmetic.

True and false are represented by #t and #f. Actually only #f is really false when a Boolean type is required, everything else will be considered true, including the empty list. Symbols can be created in at least the following ways:

'symbol

(string->symbol "symbol")

**Equality**

Scheme has three different types of equality: "eq?" returns #t if its parameters represent the same data object in memory; "eqv?" is generally the same as eq? but treats some objects (eg. characters and numbers) specially so that numbers that are = are eqv? even if they are not eq?; equal? compares data structures such as lists, vectors and strings to determine if they have congruent structure and eqv? contents.

Type dependent equivalence operations also exist in Scheme: string=?; compares two strings; char=? compares characters; = compares numbers.

**Control structures**

**Conditional evaluation**

(if test then-expr else-expr)

The test expression is evaluated, and if the evaluation result is true (anything other than #f) then the then-expr is evaluated, otherwise else-expr is evaluated.

A form that is more convenient when conditionals are nested is cond:

(cond (test1 expr1 ...)

(test2 expr2 ...)

...

(else exprn))

The first expression for which the test evaluates to true will be evaluated. If all tests result in #f, the else clause is evaluated.

A variant of the cond clause is

(cond ...

(test => expr)

...)

In this case, expr should evaluate to a function that takes one argument. If test evaluates to true, the function is called with the return value of test.

**Loops**

Loops in Scheme usually take the form of tail recursion. Scheme implementations are required to optimize tail calls so as to eliminate use of stack space where possible, so arbitrarily long loops can be executed using this technique.

A classic example is the factorial function, which can be defined non-tail-recursively:

(define (factorial n)

(if (= n 0)

1

(\* n (factorial (- n 1)))))

(factorial 5)

;; => 120

This is a direct translation of the mathematical recursive definition of the factorial: the factorial of zero (usually written *0!*) is equal to 1, while the factorial of any greater natural number *n* is defined as *n*! = *n* \* (*n* − 1)!.

However, plain recursion is by nature less efficient, since the Scheme system must maintain a stack to keep track of the returns of all the nested function calls. A tail-recursive definition is one that ensures that in the recursive case, the outermost call is one back to the top of the recurring function. In this case, we recur not on the factorial function itself, but on a helper routine with two parameters representing the state of the iteration:

(define (factorial n)

(let loop ((total 1)

(i n))

(if (= i 0)

total

(loop (\* i total) (- i 1)))))

(factorial 5)

;; => 120

A higher order function like *map*, which applies a function to every element of a list, can be defined non-tail-recursively:

(define (map f lst)

(if (null? lst)

lst

(cons (f (car lst))

(map f (cdr lst)))))

(map (lambda (x) (\* x x)) '(1 2 3 4))

;; => (1 4 9 16)

or can also be defined tail-recursively:

(define (map f lst)

(let loop ((lst lst)

(res '()))

(if (null? lst)

(reverse res)

(loop (cdr lst)

(cons (f (car lst)) res)))))

(map (lambda (x) (\* x x)) '(1 2 3 4))

;; => (1 4 9 16)

In both cases the tail-recursive version is preferable due to its decreased use of space.

For basic looping, Scheme supports a simple *do* iterator construct:

(do ((<variable1> <init1> <step1>)

...)

(<test> <expression> ...)

<command> ...)

For example:

(let ((x '(1 3 5 7 9)))

(do ((x x (cdr x))

(sum 0 (+ sum (car x))))

((null? x) sum)))

**Input/output**

Scheme has the concept of *ports* to read from or to write to. R5RS defines two default ports, accessible with the functions current-input-port and current-output-port, which correspond to the Unix notions of stdin and stdout. Most implementations also provide current-error-port.

Examples

**Hello world**

(begin

(display "Hello, World!")

(newline))

Predicate Procedures:

**procedure**: (char? *obj*)   
**returns:** #t if *obj* is a character, #f otherwise

**procedure**: (number? *obj*)   
**returns:** #t if *obj* is a number object, #f otherwise   
**procedure**: (complex? *obj*)   
**returns:** #t if *obj* is a complex number object, #f otherwise   
**procedure**: (real? *obj*)   
**returns:** #t if *obj* is a real number object, #f otherwise   
**procedure**: (rational? *obj*)   
**returns:** #t if *obj* is a rational number object, #f otherwise   
**procedure**: (integer? *obj*)   
**returns:** #t if *obj* is an integer object, #f otherwise