```
; OurScheme Intro. (modified from MIT Scheme) (version : 2011-05-11)

; Meaning vs. representation   --- on the abstract level
;
; We communicate with each other.  We communicate with the system.
;
; When we communicate, we have no choice but to use some
;   symbolic expression to convey our meaning (the "things"
;   that we mean).
;
; Each symbolic expression (if legal) ought to mean something.
; Each symbolic expression (if legal) has a MEANING.
;
; We use symbolic expressions to convey our meaning.
; The system understands our meaning.
; Sometimes, the system is asked (by us) to do what we mean.
; When the system has a result to show to us, it has no choice
;   but to show a symbolic representation of it.  Supposedly,
;   we know what the expression (what the system shows) means.



; 1. A list (or a dotted pair) is CONSTRUCTED.

> (cons 3 4)    ; an operation on two objects
(3 . 4)         ; a representation of the resulting object

> (CONS 3 4)
ERROR (unbound symbol) : CONS

; When evaluating an S-exp, Scheme/Lisp treats the first token
; after '(' as a function call

; 'cons' is a shorthand for "construct";
; implementation aspect : a CONS-cell is created

; OurScheme distinguishes between upper and lower cases
; // However, Petite Scheme does not.

> (list 3 4)    ; another operation on two objects
(3 4)          ; a representation of the resulting object

; Notice the difference between (3 . 4) and (3 4)



; 2. To "by pass" the default interpretation of an S-exp
;    by the system, use QUOTE.

> (3 4 5)
The object 3 is not applicable  ; // or "Invalid function: 3"

> '(3 4 5)
(3 4 5)

> (quote (3 4 5))
(3 4 5)

> (cons 3 (4 5))
The object 4 is not applicable  ; // or "Invalid function: 4"

> (cons 3 '(4 5))
(3 4 5)

> (list 3 '(4 5))
(3 (4 5))
```

```
> (list 3 '(4 5) 6 '(7 8))
(3 (4 5) 6 (7 8))



; 3. To give a (symbolic) name to an object

> a
ERROR (unbound symbol) : a

> (define a 5)    ; "令 a 為 5"; 讓我們把"那個東西"又稱為'a'
a defined

> a              ; is 'a' a name for something?
5               ; yes, 'a' is a name of "this thing"

> (define x '(3 4 5))   ; 讓我們把"那個東西"又稱為'x'
x defined

> x             ; Is 'x' a name for something?
(3 4 5)         ; 'x' is a name of "this thing"


// In addition, 'define' can only be called on the top level ;
// If 'define' is called on any "inner level", it will be an error ;


; 4. Whenever a function is called, its parameters are evaluated
;   first.

>(cons 3 4)
(3 . 4)

>(cons 3 b)
ERROR (unbound symbol) : b   ; or "Symbol's value as a variable is void: b"

>(cons 3 a)
(3 . 5)

>(define a '(3 4))
a defined

>(cons 5 a)
(5 3 4)



; 5. Different parts of a list (or a dotted pair) can be
;   individually accessed

> (car '(3 4))    ; CAR is used to access the "left part" of
3               ; the "starting" CONS-CELL
                ; The other way of seeing it is to say that
                ; CAR accesses the first element of a list

> (car '((3 4) 5)  )
(3 4)

> (car '((3 4) 5 . 6)  )
(3 4)

> (car '((3 4) . 5)  )
(3 4)

> (car a)
```

```
3

> (cdr '((3 4) 5)  )  ; CDR is used to access the "right part"
(5)                ; of the "starting" CONS-CELL
                  ; The other way of seeing it is to say that
                  ; CDR accesses the "remaining part" of a list

> (cdr '((3 4) 5 . 6)  )
(5 . 6)

> (cdr '((3 4) . 5)  )
5

> (cdr a)
(4)

; Different parts of a list can be accessed by mixing the use of
; CAR and CDR

> (car (cdr '((3 4) 5)   ))
5

> (car (cdr '((3 4) 5 . 6)  )
5

> (car (cdr '((3 4) 5  6  7)   ))
5

> (cdr (cdr '((3 4) 5  6  7)   ))
(6 7)




; 6. User defined functions can be created
;    To give a name to a function

> (define (f x y) (cons y x))   ; Other Lisps use
f defined                       ; DEFUN
                                ; e.g.,
                                ; (defun f (x y) (cons y x))

> (f 3 4)
(4 . 3)

> (define b '(5 6 7))
b defined

> (f a b)
((5 6 7) 3 4)

> (define (g x y)
         (* (+ x y)
            (* x y))
  )
g defined

> (define (h x y)
         (+ x y)
         (* x y)
  )
h defined

> (g 3 5)
120

> (h 3 5)
15
```

```
; 7. Primitive predicates (A predicate is a function that returns
;     "true" or "false"; By convention, the name of a predicate
;     should have a suffix '?')

> (pair? 3)     ; Other Lisps do not have PAIR
nil             ; They have ATOM, which returns the opposite
                ; logical value
                ; e.g.,
                ; > (atom 3)
                ; #t

> (pair? '(3 4))
#t             ; > (atom '(3 4))
              ; nil

> (pair? '(3 . 4))
#t

> (pair? "Hello, there!")
nil

> (null? '())    ; is it the empty list?
#t             ; yes

> (null? #f)
#t

> (null? '(3 . 4))
nil             ; no, it is not the empty list

> (integer? 3)
#t

> (integer? 3.4)
nil

> (real? 3)
nil

> (real? 3.4)
#t

> (number? 3)
#t

> (number? 3.4)
#t

> (string? "Hi")
#t

> (string? 3.4)
nil

> (boolean? #t)
#t

> (boolean? '())
#t

> (boolean? #f)
#t

> (boolean? '(3 . 4))
nil
```

```
> (symbol? 'abc)
#t

> (symbol? 3)
nil




; 8. Basic arithmetic, logical and string operations

> (+ 3 7)
10

> (- 3 7)
-4

> (- 3.2 5)
-1.7999999999999998

> (* 3 4)
12

> (define a 5)
a defined

> (/ 15 a)
3

> (/ 15.1 4)
3.775

> (/ 15.1 (+ 2 2))
3.775

> (not #t)
nil

> (not (pair? 3))
#t

> (and (pair? 3) (null? '())  )
nil

> (or (pair? 3) (null? '())  )
#t

> (> 3 2)
#t

> (> 3.1 2)
#t

> (>= 3.2 2)
#t

> (< 3.1 2)
()

> (<= 3.1 2)
nil

> (= 2 2)
#t

> (string-append "Hello," " there!")
```

```
"Hello, there!"

> (string>? "abc" "abc")
nil
```

```
; 9. eqv? and equal?

; eqv? returns "true" only when the two being compared
; objects are atoms (except in the case of strings)
; or when the two being compared objects "occupy the
; same memory space".
```

1. The two arguments of 'eqv?' OUGHT TO BE evaluated first.

2. Let the evaluated result of the first argument be ★.
   Let the evaluated result of the second argument be ☆.

3. 'eqv?' returns '#t' if
    ★ ☆ are the same atom (except in the case of strings)
    or
    ★ ☆ (including the case of strings) occupy the same memory space

4. 'eqv?' returns 'nil' otherwise.

Example :

```
> (eqv? a a)
ERROR (unbound symbol) : a

> (define a '(1 3))
a defined

> (eqv? a a)
#t

> (define b a)
b defined

> (eqv? a b)
#t

> (eqv? '(1 3) '(1 3))
nil

> (eqv? a (car (cons a '(2 3))))
#t

> (eqv? "Hello, there" "Hello, there")
nil

> (define a "Hello, there")
a defined

> (eqv? a "Hello, there")
nil

> (eqv? a (car (cons a '(2 3))))
#t

> (define a 1)
a defined

> a
1
```

```
> (eqv? a 1)
#t

> (eqv? "Hi" "Hi")
nil

> (define a 'a)
a defined

> a
a

> (eqv? a 'a)
#t

> (define b 'a)
b defined

> (eqv? a b)
#t

> (define a 'abc)
a defined

> (define b 'abc)
b defined

> (eqv? a b)
#t

> (eqv? a 'abc)
#t

; equal? corresponds the usual notion of
; equality comparison

> (equal? a a)
#t

> (equal? '(3 4) '(3 4))
#t

> (equal? "Hi" "Hi")
#t




; 10. Conditionals

> (if (> 3 2) 'good 'bad)
good

> (if a 'good 'bad)
good

> (if (not a) 'good 'bad)
bad

> (cond ((> 3 4) 'bad)
        ((> 4 3) 'good)
        (else "What happened?")  ; even though 'else' is unbound
  )
good

> (cond ((> 3 4) 'bad)
       ((> 4 5) 'bad)
```

```
        (#t "What happened?")
  )
"What happened?"

> (define else #t)
else defined

> (cond ((> 3 4) 'bad)
        ((> 4 5) 'bad)
        (else "What happened?")
  )
"What happened?"

> (cond ((> 3 4) 'bad)
       ((> 4 5) 'bad)
  )
ERROR (no return result) : cond

> (cond ((> 3 4) 'bad)
       ((> 4 3) 'good)
  )
good




; 11. Sequencing vs. functional composition

> (define (f x y) (+ (* x y) x)   )
f defined

> (f 3 5)
18

> (define d 20)
d defined

> (define (g x y) (define d (* x y)) (+ d x)  )
ERROR (define format)

> (define (g d y) (+ (* d y) d))
g defined

> (g 3 5)
18

> d
20

> (if #t 3 5)
3

> (if #t (begin 3 4 5) (begin 6 7))
5

> (if #t (3 4 5) (6 7))
ERROR (attempt to apply non-function) : 3

> (cond ((> 5 3) 'good 'better 'best) (#t 'OK?)   )
best

; Remember!  A function must always RETURN something.
; And that "value of the 'function application' " is what
; the (Lisp) system is trying to obtain.
; 12. Meaning of DEFINE revisited ("令")

; Basically, DEFINE sets up a (temporary) binding between a symbol
; and an S-expression

; However, when a "lambda expression" is evaluated, a "compiled
; function" is created internally, and the "returned value" is ...

> (lambda (x) (+ x 5) )    ; a function is described; it has no name.
#function

; DEFINE sets up the binding between a name and (in the case of
; lambda expressions) the internal definition of a function

> (define f (lambda (x) (+ x 5))  )
f defined

> (f 3)
8

> ((lambda (x) (+ x 5)) 3)
8

> (define g '(lambda (x) (+ x 5))  )
g defined

> (g 3)
The object (lambda (x) (+ x 5)) is not applicable.
```

```
; 13. Local variables

; Use LET to create (local) symbol bindings ("令")

> (let ( (x 5)
         (foo (lambda (y) (bar x y)))
         (bar (lambda (a b) (+ (* a b) a)))
       )
      (foo (+ x 3))
 )
45

> (define (f z)
   (let ( (x 5)
         (foo (lambda (y) (bar x y)))
         (bar (lambda (a b) (+ (* a b) a)))
       )
       (+ (foo (+ x 3))
         z)
  ))
f defined

> (f 7)
52

> (foo 2)
ERROR (unbound symbol) : foo

> (let ((x 5) (y 6) (z 7)
        (foo (lambda (y) (bar x y)))
        ( bar (lambda (a b) (+ (* a b) a)))
       )
      (+ (foo (+ x 3)) y z)
 )
58
```

```
; 14. Change of (local) symbol bindings ("assignment")

; Assignments are of the form 'set...!'
; e.g.,
; set!   set-car!   set-cdr!

> (set! w 7)
7

> w
7

> (define a '(3 4))
a defined

> (define b a)
b defined

> (set-car! a 5)
ERROR (no return result) : set-car!

> a
(5 4)

> b
(5 4)

> (set-cdr! a 7)
ERROR (no return result) : set-cdr!

> a
(5 . 7)




; 15. Input and output
; four output functions : write, write-line, display-string, newline
; these three functions do not have return-values
; two input functions : read, read-line

> (write "Hi")
"Hi"ERROR (no return result) : write

> (write (+ 3.4 5))
8.4ERROR (no return result) : write

> (begin (write 5) (newline) (write "Hi"))
5
"Hi"ERROR (no return result) : write

> (begin (write-line 5) (write-line 7) 9)
5
7
9

> (display-string "Hi")
HiERROR (no return result) : display-string

> (begin (display-string "Please enter: ") (read))
Please enter: 5
5

> (define x (begin (display-string "Please enter: ") (read)
          ))
```

```
Please enter: (4 a)
(4 a)

> x
(4 a)

> (define y (begin (display-string "Please enter: ") (read)
          ))
Please enter: Hi, there!
y defined

> ERROR (unbound symbol) : there!

> y
Hi,

> (define z (begin (write-string "Please enter: ") (read)
          ))
Please enter: "Hi, there!"
z defined

> z
"Hi, there!"

> (define w (begin (write-string "Please enter: ") (read-line)
          ))
Please enter: Hi, there!
w defined

> w
"Hi, there!"

> (define h (begin (write-string "Please enter: ") (read-line)
          ))
Please enter: 5
h defined

> h
"5"
```

```
; 16. Load files

> (load "file1.txt")
Loading "file1.txt" -- done
ERROR (no return result) : load

> (make-directory "TestDir")
ERROR (no return result) : make-directory

> (load "TestDir/file1.txt")
Loading "testdir/file1.txt" -- done
ERROR (no return result) : load
```

```
; 17. Static scoping

> (define a 10)
a defined

> (define (f x) (+ x a))   ; Which 'a' is this?
f defined

> (define (g a) (+ a (f 5))) ; Which 'a' is this?
```

```
g defined

> (g 30)
45                 ; 65 if dynamic scoping is used




; 18. Eval

;    Whatever the evaluated result of the first argument is,
;    it must be expressed in the form of an S-expression.
;    'eval' takes this S-expression and evaluate it.

> (define a '(1 2 3))
a defined

> (car a)
1

> (eval '(car a))
1

> (eval '(car '(1 2 3)))
1

> (eval '(1 2 3) )
ERROR (attempt to apply non-function) : 1

> (define a '(car '(1 2 3 4)))
a defined

> a
(car (quote (1 2 3 4)))

> (eval a)
1
```