```
// ==========================================================================

How to write OurScheme (Latest modification : 05/31, 2011)

// ==========================================================================
```

Main program for Project 1

```
  Print 'Welcome to OurScheme!'

  repeat

    Print '> '

    ReadSExp(exp);

    if no error
      then PrintSExp(exp);
    else
      PrintErrorMessage() ;

  until (OR (user entered '(exit)')
            (END-OF-FILE encountered)
        )

  Print 'Thanks for using OurScheme!' or EOF error message
```

Main program for the remaining projects

```
  Print 'Welcome to OurScheme!'

  repeat

    Print : '> '

    ReadSExp( s_exp );

    if no error
```

```
    then result <- EvalSExp( s_exp );
         if error
           PrintErrorMessage();
         else
           PrintSExp( result ) ;
  else PrintErrorMessage() ;

until user has just entered LEFT_PAREN "exit" RIGHT_PAREN
       or
       EOF encountered

Print 'Thanks for using OurScheme!' or EOF error message
```

一、 Read in an S-expression

First, try to read in an S-expression

terminal :

```
LEFT-PAREN  // '('
RIGHT-PAREN // ')'
INT         // e.g. '123', '+123', '-123'
STRING      // "string's (example)."
            // (strings do not extend across lines)
DOT         // '.'
FLOAT       // '123.567', '123.', '.567', '+123.4', '-.123'
NIL         // 'nil' or '#f', but not 'NIL' nor 'nIL'
T           // 't' or '#t', but not 'T' nor '#T'
QUOTE       // '
SYMBOL      // a consecutive sequence of printable characters
            // that are not numbers,
            // and do not contain '(', ')',
            // single-quote, double-quote and white-spaces ;
            // Symbols are case-sensitive
            // (i.e., uppercase and lowercase are different);
```

Note :

With the exception of strings, token are separated by the following "separators" :

(a) one or more white-spaces

(b) '('

(note : '(' is a token by itself)

(c) ')'

(note : ')' is a token by itself)

(d) the single-quote character (')

(note : it is a token by itself)

(e) the double-quote character (")

(note : it starts a STRING)

Examples :

'3.25' is a FLOAT.

'3.25a' is a SYMBOL.

'a.b' is a SYMBOL.

'#f' is NIL

'#fa' (alternatively, 'a#f') is a SYMBOL.

Note :

'.' can mean several things :

it is either part of a FLOAT or part of a SYMBOL or a DOT.

It means a DOT only when it "stands alone".

'#' can also mean two things :

it is either part of NIL (or T) or part of a SYMBOL.

It is part of NIL (or T) only when it is '#t' or '#f' that

"stand alone".

```
<S-exp> ::= <ATOM>
            | LEFT-PAREN <S-exp> { <S-exp> } [ DOT <S-exp> ]
              RIGHT-PAREN
            | QUOTE <S-exp>
```

```
<ATOM>  ::= SYMBOL | INT | FLOAT | STRING
            | NIL | T | LEFT-PAREN RIGHT-PAREN
```

Once the attempt to read in an S-expression fails, the line containing the error-char is ignored. Start to read in an S-expression from the next input line.

```
> (t . nil . (1 2 3))
ERROR (unexpected character) : line 1 column 10 character '.'

> (12 (    . 3))
ERROR (unexpected character) : line 1 column 11 character ' '

> ())
nil
> ERROR (unexpected character) : line 1 column 1 character ')'

> (1 2 3) )
( 1
  2
  3
)

> ERROR (unexpected character) : line 1 column 2 character ')'

>'(1 2 3) )
( quote
  ( 1
    2
    3
  )
)

> ERROR (unexpected character) : line 1 column 2 character ')'
```

二、 Always check the syntax of the user's input; Must make sure that it is an S-expression before evaluating it.

User input 可能會有的三種 syntax error 的相關 message(的範例)如下：

ERROR (unexpected character) : line 1 column 2 character ')'

ERROR (unexpected character) : line 3 column 27 LINE-ENTER encountered

ERROR : END-OF-FILE encountered when there should be more input

三、 The part of eval() concerning error messages : // Note : once an error occurs,
// the call to eval()
is over

if what is being evaluated is an atom but not a symbol

return that atom

else if what is being evaluated is a symbol

check whether it is bound to an S-expression or an internal function

if unbound

ERROR (unbound symbol) : abc

else

return that S-expression or internal function (i.e., its binding)

else // what is being evaluated is (...) ; we call it *the main S-expression* below

// this (...) cannot be nil (nil is an atom)

if (...) is not a (pure) list

ERROR (non-list) : (...) // (...)要pretty print

else if first argument of (...) is an atom ☆, which is not a symbol

ERROR (attempt to apply non-function) : ☆

else if first argument of (...) is a symbol SYM

check whether SYM is the name of a function (i.e., check whether 「SYM has a
binding, and that binding is an internal
function」)

if SYM is the name of a known function

```
      if the current level is not the top level, and SYM is 'clean-environment' or
         or 'define' or  'exit'

        ERROR (clean-environment format) / ERROR (define format) / ERROR (level of
exit)
        // Project 2 的test data規定要 ERROR (clean-environment/define format)，暫不改它。

      if SYM is 'define' or 'let' or 'cond' or 'lambda'

        check the format of this expression // 注意：此時尚未check num-of-arg
        // (define symbol    // 注意：只能define 非primitive的symbol (這是final
decision!)
        //           S-expression
        // )
        // (define ( one-or-more-symbols )
        //           one-or-more-S-expressions
        // )
        // (lambda (zero-or-more-symbols)
        //           one-or-more-S-expression
        // )
        // (let (zero-or-more-PAIRs)
        //           one-or-more-S-expressions
        // )
        // (cond one-or-more-AT-LEAST-DOUBLETONs
        // )
        // where PAIR df= ( symbol S-expression )
        //        AT-LEAST-DOUBLETON df= a list of two or more S-expressions

        if format error
          ERROR (cond parameter error) : 出問題的(照理應是個AT-LEAST-DOUBLETON的)S-exp
          or
          ERROR (define format)       // 之所以這四個message會有不同的格式、是因為之前在已做之
          or                              // projects的test data之中已有規定、無法再改。
          ERROR (LET format)           //
          or                              // 目前就維持不同的格式。要改、以後(e.g., PL993)再改。
          ERROR (lambda format)       // (真要改的話，未見得是誰向誰看齊...)

        evaluate ( ... ) // for 'cond', there may be ERROR (no return value) : cond
```

```
      return the evaluated result (and exit this call to eval())
    else // SYM is a known function name 'abc', which is neither
         // 'define' nor 'let' nor 'cond' nor 'lambda'

      check whether the number of arguments is correct

      if number of arguments is NOT correct
        ERROR (incorrect number of arguments) : abc

  else // SYM is 'abc', which is not the name of a known function

    ERROR (unbound symbol) : abc
    or
    ERROR (attempt to apply non-function) : ☆ // ☆ is the binding of abc

else // the first argument of ( ... ) is ( ∘ ∘ ∘ ), i.e., it is ( ( ∘ ∘ ∘ ) ..... )

  evaluate ( ∘ ∘ ∘ )

  // if any error occurs during the evaluation of ( ∘ ∘ ∘ ), we just output an
  // an appropriate error message, and we will not proceed any further

  if no error occurs during the evaluation of ( ∘ ∘ ∘ )

    check whether the evaluated result (of ( ∘ ∘ ∘ )) is an internal function

    if the evaluated result (of ( ∘ ∘ ∘ )) is an internal function

      check whether the number of arguments is correct

      if num-of-arguments is NOT correct
        ERROR (incorrect number of arguments) : name-of-the-function
        or
        ERROR (incorrect number of arguments) : lambda expression
                                                // in the case of nameless
functions
```

```
      else // the evaluated result (of ( ◦ ◦ ◦ )) is not an internal function
        ERROR (attempt to apply non-function) : ☆ //  ☆ is the evaluated result

  eval the second argument S2 of (the main S-expression) ( ... )

  if the type of the evaluated result is not correct
    ERROR (xxx with incorrect argument type) : the-evaluated-result
    // xxx must be the name of some primitive function!

  if no error
    eval the third argument S3 of (the main S-expression) ( ... )

  if the type of the evaluated result is not correct
    ERROR (xxx with incorrect argument type) : the-evaluated-result

  ...

  if no error
    apply the binding of the first argument (an internal function) to S2-eval-result,
    S3-eval-result, ...
    if no error
      if there is an evaluated result to be returned
        return the evaluated result
      else
        ERROR (no return result) : name-of-this-function
        or
        ERROR (no return result) : lambda expression // if there is such a case ...
```

end // else what is being evaluated is (...) ; we call it *the main S-expression*

Note :

1. error message之「其他」

如果你的系統碰到一個error、而以上eval的algorithm中對此error「該有何error message」並沒有規範(這有點像是if-then-else-if-then-...-else-if-then-else中的最後那個「else」)，你就output

ERROR : aaa

其中 aaa 是 user input 中「出問題的那個「被 evaluate 的 function」的 first argument」。

當你依照以上 eval 的 algorithm 來 evaluate an S-expression 時，你會不斷的要 evaluate a function，一旦這種「project 並未規範的 error」發生，「當時」那個被 evaluate 的 function 的 first argument 就是這裡所謂的 aaa。

// 「project 未規範」 df= project 中(與 test data 中)未提到這種 error，但明明就是個 error
// | // i.e., OR
// project 中有提到這種 error，但沒說 error message 應該是啥

e.g.,

> (/ a 0)
ERROR : /

2. Some examples of error messages

```
> (car nil)
ERROR (car with incorrect argument type) : nil

> (define (f a) (cons a a))
f defined

> (f 5)
( 5
  .
  5
)

> (f 5 a)
ERROR (incorrect number of arguments) : f

> (define (ff a) (g a a))
ff defined
```

```
> (define (g a) (cons a a))
g defined

> (ff 5)
ERROR (incorrect number of arguments) : g

> (define (f a) (cons a a a))
f defined

> (f 5)
ERROR (incorrect number of arguments) : cons

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

> (cons hello 4)
ERROR (unbound symbol) : hello

> hello
ERROR (unbound symbol) : hello

> (CONS hello there)
ERROR (unbound symbol) : CONS

> (cons 1 2 3)
ERROR (incorrect number of arguments) : cons

> (3 4 5)
ERROR (attempt to apply non-function) : 3

> (cons 3
        (4321 5))
ERROR (attempt to apply non-function) : 4321

> (define a 5)
5

> (a 3 a)
```

```
ERROR (attempt to apply non-function) : 5


> (* 3 "Hi")
ERROR (* with incorrect argument type) : "Hi"


> (string>? 15 "hi")
ERROR (string>? with incorrect argument type) : 15


> (+ 15 "hi")
ERROR (+ with incorrect argument type) : "hi"


> (string>? "hi" "there" a)
ERROR (string>? with incorrect argument type) : 5


> (string>? "hi" "there" about)
ERROR (unbound symbol) : about


> (cond ((> 3 4) 'bad)
        ((> 4 5) 'bad)
  )
ERROR (return value undefined) : cond


> (cond ((> y 4) 'bad)
        ((> 4 3) 'good)
  )
ERROR (unbound symbol) : y
```


3.value and binding

Lisp and Scheme 堅持一個概念：

沒有「value」！ 只有「binding」！

也就是說：

沒有「symbol 的 value」這回事！ 只有「symbol 的 binding」！

* Symbol 的 binding 可能是一個 S-expression (which is basically a structure of symbols)，也可能是一個(所謂的)internal function。

* Internal functions 有事先 system define 好的，也有 user define 的。

* evaluate 一個「非 symbol 的 atom」 的結果 是 那個 atom

* evaluate 一個 symbol 的結果 是 那個 symbol 的 binding

* evaluate 一個 list 的結果 是 apply 「evaluate 此 list 的 first argument 所得的結果」(which is supposedly an internal function) 於 「evaluate 此 list 的其他 argument 所得的結果」

經由使用某些 system defined 的"東東" (如'define')，我們可以改變 symbol 的 binding。

但 我們能改變「原先 system 已 define 好的 symbol」的 binding 嗎？

例： how about these？


```
> (define define 3)
???
```



```
> (define exit 3)
???
```



```
> (let ((cons car)) (cons '(1 2)))
???
```

Petite Scheme 允許如此！ OurScheme 要不要？

答案： 我們 不 允許改變"primitive symbol"的 binding！！！

```
// 有人曾問這行不行： (define 3 4)
// 答案固然是不行，但原因是： 'define'只能改變「symbol」的 binding

// 也有人曾問這行不行： (define nil 4)
// 答案固然是不行，但原因是： 'nil'不是「symbol」
```

4. Expected argument type

Below, the word 'symbol' should be taken to mean : a symbol that

is not a primitive symbol (i.e., it is not a pre-defined symbol)

* 'car' expects its argument to be a cons-cell.

* 'cdr' expects its argument to be a cons-cell.

* 'quote' expects its argument to be an S-expression.

* 'define' expects that either its first argument is a symbol or its first argument is a list of one or more symbols.

* 'lambda' expects that its first argument is a list of zero or more symbols.

* 'let' expects its first argument to be a list of one or more pairs, with the first element of each pair being a symbol.

* '+', '-','*', '/' expect their arguments to be numbers.

* '>', '>=','<','<=' expect their arguments to be numbers.

* 'string-append' and 'string>?' expect their arguments to be strings.

* 'set!', 'set-car!' and 'set-cdr!' expect their first argument to be a symbol.

* 'display-string' expects its argument to be a string.

* 'load' and 'make-directory' expect their arguments to be strings.

* In all other cases, S-expressions or internal functions are expected as arguments.