2017 spring, PL project 4 (OurScheme Project 4)

Due : 6/25, 2017(Sunday) before midnight

// ===

For this project, you need to extend your OurScheme interpreter, so that
it accepts the following function calls : create-error-object, error-object?,
read, write, display-string, newline, eval and set!.

Below is a description of these functions.

    create-error-object
    error-object?

        Add ERROR to your list of allowable data types, so that
        there are not only INT, FLOAT, STRING, etc. but also ERROR.

        'create-error-object', a function that you should add to your system,
        accepts one argument which is a string, and returns an error-object.

        'error-object?' is a boolean, one-argument function that tests whether
        the given argument is an error-object.

        An error-object "contains" a string (an appropriately phrased error
        message), and "behaves" just like a string-object. There are only two
        differences between an error-object and a string-object. One, while
        'error-object?' returns #t in the case of an error-object, it returns
        nil in the case of a string-object. Two, while string-objects can be
        read in, error-objects cannot. Error-objects can only be explicitly
        created by calling 'create-error-object'. Calling the function 'read'
        may also result in an error object to be implicitly created, as
        described below.

    read

        'read' is the main input function. It attempts to first read in the next
        input S-expression and then return (the internal representation of) that
        S-expression. 'read' does not accept any argument.

        If there is any input error, 'read' returns an error-object.
        The error message "contained" in this error-object will be the same as

what ReadSExpression() would print or return if it is ReadSExpression() (and not 'read') that is doing the input.   (However, there will be no "terminating line-enter character".)   Note that if the input error 'read' encounters is end-of-file, the error-object that 'read' returns will contain the string :

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

Quick summary :

    'read', when called, returns one of two "things" :
        (the internal representation of) a normal S-expression, or
        an error-object that "contains" a string (an error message).

    An error-object behaves just like a string object.   It "contains"
    an error message.   We can use 'error-object?' to see whether
    'read' returns an error-object.   And if what 'read' returns is
    an error-object, we can also compare that error-object with
    "ERROR : END-OF-FILE encountered when there should be more input"
    to see whether the input error that 'read' encountered was an
    end-of-file error.

Example :

```
> (define a (read)) (1 2 3)
a defined

> (error-object? a)
nil

> a
( 1
  2
  3
)

> (define a (read))( 1 3 5 . 7 s )
a defined

> (error-object? a)
#t
```

```
> (equal?
    a
    "ERROR : END-OF-FILE encountered when there should be more input")
nil

> a
"ERROR (unexpected character) : line 1 column 13 character 's'"

; Below cannot happen in the case of OurScheme, because we use the
; same input stream for getting user input
; (See 『中文補充說明之三：「哪個 input 字元是誰讀的？」』 below) ;
; But if 'read' gets its input from some other place, the following
; can happen ;
;
; > (define b (read)) ; suppose it is EOF in that "other place"
; b defined
;
; > (error-object? b)
; #t
;
; > (equal?
;      b
;      "ERROR : END-OF-FILE encountered when there should be more input")
; #t
;
; > b
; "ERROR : END-OF-FILE encountered when there should be more input"
;
; > (define c (read))
; c defined
;
; > (equal?
;      c
;      "ERROR : END-OF-FILE encountered when there should be more input")
; #t
;
; > c
; "ERROR : END-OF-FILE encountered when there should be more input"
```

When 'read' encounters an input error, it works in exactly the same way as how ReadSExpression() works. For a detailed explanation of this, see 「中文補充說明之二：'read'所應回傳的 error message」 below.

write
display-string
newline
symbol->string
number->string

There are three output functions: write, display-string, newline.

'write' accepts one argument, and the argument should be (the internal representation of) a legitimate S-expression. 'write' prints the given argument in a way that is just like how PrintSExpression() works.

While 'write' prints any STRING object with enclosing quotes printed, display-string prints the same STRING object without printing the two enclosing quotes.

display-string 的(唯一)參數只能是 string 與 error-object
// 因為名字叫'display-string'的關係

'newline' just prints a line-enter character in the output.

As for the return-values, 'write' returns its argument as the return-value, 'display-string' also returns its argument as the return-value, whereas 'newline' just returns nil.

'symbol->string' accepts a symbol and returns an S-expression that is just a string.

'number->string' accepts a number and returns an S-expression that is this number "in string format." Examples : (number->string 23) should return "23", whereas (number->string 23.0) should return "23.000". (If the number is not an integer, then just use printf("%.3f", ...) of String.format("%.3f", ...) to get the string form of it).

Example :

```
> (write '(1 2 3))
( 1
  2
  3
)( 1
```

```
    2
    3
)

> (write "hi")
"hi""hi"

> (display-string "hi")
hi"hi"

> (display-string (write "hi"))
"hi"hi"hi"

> (write (display-string "hi"))
hi"hi""hi"

> (newline)

nil

> (begin (write "hi") (newline) (display-string "hi") (newline))
"hi"
hi
nil

> (define a "hello")
a defined

> (write (display-string a))
hello"hello""hello"

> (symbol->string 'Hi)
"Hi"

> (display-string (symbol->string 'Hi))
Hi"Hi"

> (string-append (symbol->string 'Hi) " is a good thing")
"Hi is a good thing"

> (string-append (number->string 45) " is a number")
"45 is a number"
```

```
    > (string-append (number->string 4.561857) " is also a number")
    "4.562 is also a number"

    > (string-append "Just as " (number->string 4.561357) " is a number")
    "Just as 4.561 is a number"

    >
```

eval

'eval', which accepts one argument, evaluates the given (internal representation of an) S-expression.

Internally, 'eval' just amounts to calling EvalSExpression().

Example :

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

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

> (eval "Hi")
"Hi"

> (eval '(fun '(1 2 3)))
ERROR (unbound symbol) : fun

> (eval (fun '(1 2 3)))
ERROR (unbound symbol) : fun

> (eval (car '( (cdr '(1 2 3))
                        5
                      )
                )
          )
  ( 2
    3
  )
```

set!

除了'define', 'clean-environment', 'car', 'cdr', 'cons' 等 primitive functions
之外，要再加一個'set!'指令。

'set!'基本上是另一個版本的'define'，它接受兩個參數，第一個參數應該是個 symbol，
不 evaluate；第二個參數是任一個 S-expression，要 evaluate。

以'(set! a '(1 2 3))'為例，這指令會把'a'這個 symbol 的 binding 設為'(1 2 3)'這個
list。

'set!'與'define'的主要不同之處在於：

(1) 'set!'有 return 值，而且此 return 值「被 set 的那個值」(此思維是來自 C)。
(2) 'set!'可以出現在一個(我們要 evaluate 的)S-expression 的任一 level
    ('define'只能出現在一個(我們要 evaluate 的)S-expression 的 top level)

例：

```
  > (set! a '(1 2 3))
  ( 1
    2
    3
  )

  > a
  ( 1
    2
    3
  )

  > (cons (set! a '(4 5))
          a
    )
```

```
( ( 4
      5
  )
  4
  5
)

> a
( 4
  5
)

> ( cons a
         ( cons ( set! a '( 7 8 ) )
                a
         )
  ) ; See HowToWriteOurScheme.doc to know about "order of evaluation"
( ( 4
      5
  )
  ( 7
      8
  )
  7
  8
)

> (set! aaa 5) ; here is an example of how 'set!' deals with local-var and global-var
5

> (define (f aaa) (set! aaa (+ aaa 40)) (set! bbb 100) (+ aaa bbb))
f defined

> (f 10)
150

> aaa
5

> bbb
100
```

```
        >

// ================================================================================
//
// A slightly more complicated example :
//
Welcome to OurScheme!

>
(define (Factorial1 n)
    ( if (= n 0)
              1
              (* n (Factorial1 (- n 1)))
    )
)
Factorial1 defined

>
(define (Factorial n)
    (cond ( (< n 0) "Error! n less than 0."
              )
              ( (not (integer? n)) "Error! n not an integer."
              )
              ( #t (Factorial1 n)
              )
    )
)
Factorial defined

> (Factorial 5.5)
"Error! n not an integer."

> (Factorial -3)
"Error! n less than 0."

> (Factorial 5)
120

> (define (Test1)
        (begin (display-string "Please enter a positive integer : ")
                      (Factorial (read))
        )
    )
Test1 defined

> (Test1)
Please enter a positive integer : 8.75
"Error! n not an integer."

> (Test1)
Please enter a positive integer : -.375
"Error! n less than 0."

> (Test 1)
ERROR (unbound symbol) : Test

> (Test1)
Please enter a positive integer : 9
362880

> (Test1)
Please enter a positive integer : (1 2 3)
ERROR (< with incorrect argument type) : ( 1
    2
    3
)

> (define (Factorial n)
        (cond ( (not (integer? n)) "Error! n not an integer."
                    )
                    ( (< n 0) "Error! n less than 0."
                    )
                    ( #t (Factorial1 n)
                    )
        )
    )
Factorial defined

> (Test1)
Please enter a positive integer : (1 2 3)
"Error! n not an integer."

> (define (Factorial n)
        (cond ( (not (integer? n))
                        (display-string "Parameter to 'Factorial' : ")
                        (write n)
                        (newline)
                        "Error! Not an integer!"
                    )
                    ( (< n 0)
                        (display-string "Parameter to 'Factorial' : ")
                        (write n)
                        (newline)
                        "Error! Less than 0!"
                    )
                    ( #t (Factorial1 n)
                    )
            )
        )
Factorial defined

> (Factorial 8.75)
Parameter to 'Factorial' : 8.750
"Error! Not an integer!"

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

> (Factorial '(1 2 3))
Parameter to 'Factorial' : ( 1
    2
    3
)
"Error! Not an integer!"

> (Test1)
Please enter a positive integer : (1 2 3)
Parameter to 'Factorial' : ( 1
    2
    3
)
"Error! Not an integer!"

> (eval (read))
(car '(3 4 5))
3
```

```
> (eval (read))    (car '(3 4 5))
3

> (eval (begin (display-string "Please enter an S-expression to evaluate : ")
               (read)
        )
  )
Please enter an S-expression to evaluate :
(cons 1
      '(2 3)
)
( 1
  2
  3
)

>
(define (Factorial n)
  (cond ( (not (integer? n))
          ( create-error-object "Error! parameter not an integer." )
        )
        ( (< n 0)
          ( create-error-object "Error! parameter less than 0." )
        )
        ( #t
          ( if (= n 0)
               1
               (* n (Factorial (- n 1)))
          )
        )
  )
)
Factorial defined

>
(define (Factorial_ n)
  ( let ( ( answer ( Factorial n )
          )
        )
        ( if ( error-object? answer )
             (display-string answer)
```

```
                        answer
                ) ; if
    ) ; let
) ; define
Factorial_ defined

> (Factorial_ 5.5)
Error! parameter not an integer."Error! parameter not an integer."

> (Factorial_ -3)
Error! parameter less than 0."Error! parameter less than 0."

> (Factorial_ 5)
120

>
    ( define (Test1)
        (begin (display-string "Please enter a positive integer : ")
                      ( let ( ( input ( read ) )
                                  )
                                  ( if ( error-object? input )
                                          input
                                          ( Factorial input )
                                  ) ; if
                      ) ; let
        ) ; begin
    ) ; define
Test1 defined

> (Test1)
Please enter a positive integer : 8.75
"Error! parameter not an integer."

> (Test1)
Please enter a positive integer : -.375
"Error! parameter not an integer."

> (Test1)
Please enter a positive integer : -375
"Error! parameter less than 0."

> (Test1)
```

```
Please enter a positive integer : 9
362880

> (Test1)
Please enter a positive integer : (1 2 3)
"Error! parameter not an integer."

> (Test1)
Please enter a positive integer :
( 1
  2
  .
  3
  .
  4
)
"ERROR (unexpected character) : line 6 column 3 character '.'"

> 4

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

> (exit)

Thanks for using OurScheme!
```

// ===

補充說明之一：到底要做啥？

基本上，你要 implement 四個 target functions : read, write, eval, set!。

但光是 write 並不符合 output 的需求，所以還要有 display-string 與 newline；

同時，有時有在 string 之中加入 symbol-name 與 number 的需求，所以還需要 symbol->string 與 number->string。

最後，由於加入了 error-object 的概念，所以也要有 create-error-object 與 error-object?來配合運作。

算起來總共是要 implement 十個 functions。

read 基本上只是呼叫 ReadSExpression()，write 基本上只是呼叫 PrintSExpression()，

而 eval 基本上也只是呼叫 EvalSExpression()。set!則是 define 的"孿生兄弟"。

如果 write 能直接用 PrintSExpression()來 implement，那 display-string、newline、

symbol->string 與 number->string 也應該能手到擒來。至於 error-object?

與 create-error-object，那應該佔不了多少程式碼。

也就是說，如果你的程式架構配置妥當，你將不需要加太多程式碼。

補充說明之二：'read'所應回傳的 error message

※ 基本上，(input) error 的發生、在於'read'讀到不該出現的 token (i.e., syntactic error)。

當這狀況發生時，'read'應該知道此 token(的 first 字元)的 line 與 column，

從而產生 error message 如 ：

```
ERROR (unexpected token) : atom or '(' expected when token at Line X Column Y is >>...<<
ERROR (unexpected token) : ')' expected when token at Line X Column Y is >>...<<
```

但有一個狀況比較特殊，那就是 string 還沒"close"就已出現 LINE-ENTER(這是個 lexical error)。

例： // 假設'('是 column 1 第一行共 18 字元(不計 LINE-ENTER)

```
( cons "Hi" "How
         are you" )
```

此時 error message 是 ：

```
ERROR (no closing quote) : END-OF-LINE encountered at Line 1 Column 19
```

※ 一旦發生以上所述錯誤，該行就完全 skip，系統會把使用者的下一行視為之後的 input。

※ 由於是用 input 檔給 input、而不是 interactive I/O，所以有可能發生「使用者

還沒 input 完一個 S-expression 就沒 input 了」的狀況，

此時 error message 是 ：

```
ERROR (no more input) : END-OF-FILE encountered
```

補充說明之三：「哪個 input 字元是誰讀的？」

當系統(你的程式)在讀 input 時，有兩種可能：

可能性之一：

系統剛印了'> '，現在要讀入一個 S-expression。

當此 S-expression 被讀入之後，系統就會 evaluate 這個 S-expression，

並把所 evaluate 之結果印出來。

可能性之二：

系統(因為 evaluate 某段程式碼的關係、目前)正在執行'read'，所以現在要讀

一個 S-expression。此 S-expression 被讀入之後，就會被當作是系統 evaluate

'read'所得之結果。

至於這個「系統 evaluate 'read'所得之結果」在之後會被怎麼樣，這要看程式

是怎麼寫的。

如果程式是'(define a (read))'，那麼「系統 evaluate 'read'所得之結果」

(亦即被讀入之 S-expression)就會被設定為'a'這個 symbol 的 binding。

如果程式是'(eval (read))'，那麼「系統 evaluate 'read' 所得之結果」(亦即

被讀入之 S-expression)就會被當作是'eval'的參數(而被 evaluate)。 也就是說，

```
(eval (read))(car '(1 2 3))
```

與

```
(car '(1 2 3))
```

會得到同樣的結果。

由於 OurScheme 有很清楚的 token-separators 的觀念，所以我們可以辨別

「一個 input 字元到底是在『可能性之一』被讀入還是在『可能性之二』被讀入」。

原則上，系統在讀入任一個 S-expression 之後(不管是『可能性之一』還是

『可能性之二』)，下一次要讀入一個 S-expression 時、就是從「上一個

S-expression 的最後一個字元」的下一個字元開始 process。

假設使用者在'> '之後 enter :

```
(eval (read))(car '(1 2 3))(define b 5)
```

那麼系統所讀入(並要 evaluate)的 S-expression 就是'(eval (read))'，而

當'(read)'被系統 evaluate 時，所讀入的 S-expression 就是從'(car ...'的

'('開始 process、從而導致'(car '(1 2 3))'被讀入，而下一個會被讀入

的 S-expression 將會是'(define b 5)'。

假設使用者在'> '之後 enter :

```
(eval (read))    (car '(1 2 3))    (define b 5)
```

那麼當'(read)'被系統 evaluate 時，所讀入的 S-expression 就是從

'(eval (read))'的最後一個')'之後的 SPACE 字元開始 process，

從而導致'(car '(1 2 3))'被讀入；而下一個會被讀入

的 S-expression 依舊會是'(define b 5)'(而且是從'(car '(1 2 3))'的最後

一個')'之後的 SPACE 字元開始 process)。

補充說明之四：more on error-object

基本上，一個 error-object 內含一個 string >>"..."<<。
To 'write' this error-object, output is >>"..."<<。
To 'display-string' this error-object, output is >>...<<。

若在 prompt level evaluate 一個 S-expression 而得一 error-object、
此 error-object 是以'write'的方式印出來。

例：

```
> (define a (create-error-object "This is an error!"))
a defined

> (error-object? a)
#t

> a
"This is an error!"

> (begin (write a) 5)
"This is an error!"5

> (begin (display-string a) 5)
This is an error!5

> (write a)
"This is an error!""This is an error!"

> (display-string a)
This is an error!"This is an error!"

> (car a)
ERROR (car with incorrect argument type) : "This is an error!"
```

>