PL OurScheme project for the spring of 2017, Part 2 ("Project 2")

Due : 6/25(日) midnight (23:59)

// Some test input of Project 1 may again appear in Project 2
// e.g., if in Project 1 the input was : (1 2 3)
// then in Project 2, this input may reappear as : '(1 2 3)
// or : (quote (1 2 3))

In Project 1, you have done the following :

* You wrote a scanner or a 「scanner layer」 (consisting of several functions).

  This scanner is responsible for (1) using separators to get tokens from the user's actual input, and (2) deciding what tokens they are ;

  In a sense, the scanner "transforms" the actual input stream of characters into a (conceptual) input stream of tokens.

* You wrote a parser or a 「parser layer」(consisting of several functions).

  The parser "reads" the conceptual input stream of tokens by calling the scanner, and decides whether the input stream of tokens satisfies the grammar of an S-expression.

  Once the parser makes sure that the tokens "read" satisfies the grammar of an S-expression, it constructs an internal, tree-like data structure for this S-expression.

* You wrote a "pretty-printer". Given a pointer to an internal data structure that corresponds to some S-expression, the pretty-printer prints out this S-expression in some pre-determined format.

* You managed to print an error message signifying the location of the so-called "error character" whenever the system encounters a syntax error in the user's input.

* You organized the working of your OurScheme interpreter in some way, so that the working of your system corresponds to the following "code skeleton" :

```
Print : 'Welcome to OurScheme!'

repeat

  Print : '> '

  ReadSExp( s_exp );

  if no syntax error (no "unexpected token" or "unclosed string")
    then PrintSExp( s_exp );
  else
    PrintErrorMessage() ;

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

if ( END-OF-FILE encountered ) // and NOT 「user entered '(exit)'」
  Print 'ERROR (no more input) : END-OF-FILE encountered'

Print '\n'
Print : 'Thanks for using OurScheme!'
```

For Project 2, you are to extend your system so that the following are realized (by your system).

- All "primitive exressions" (expressions that involve primitive operations) can be evaluated.
- 'define' is supported (but no definition (and use) of functions yet)
- "Conditional processing" (via the use of 'if' and 'cond') is supported.
- Sequencing (functional composition and the use of 'begin') is supported.

Your main program should now look something like the following.

```
Print : 'Welcome to OurScheme!'

repeat
```

```
    Print : '> '

    ReadSExp( inSExp );

    if no syntax error

      then

        EvalSExp( inSExp, resultSExp ) ;

        if evaluation error
          then PrintEvaluationError() ;

        else // no evaluation error
          PrintSExp( resultSExp );

      end-then // no syntax error

    else // syntax error
      PrintSyntaxError() ;

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

  if ( END-OF-FILE encountered ) // and NOT 「user entered '(exit)'」
    Print 'ERROR (no more input) : END-OF-FILE encountered'

  Print '\n'
  Print : 'Thanks for using OurScheme!'
```

!!
error 的判斷(怎樣的寫法應該算什麼樣的 error？)是以 HowToWriteOurScheme.doc 為準
!!

Below are the primitives (and features) that your system should implement.

(括號內的數字指的是這個 function 可接受的 argument 的數目 - i.e., the number of arguments that this function can take)

1. Constructors

cons (2)
list (>= 0)

2. Bypassing the default evaluation

quote (1)
' (1)

3. The binding of a symbol to an S-expression

define (2)

; Once a symbol is defined (or "bound"), the user can enter
; this symbol, and the system will return its binding.

; however, the user is not allowed to redefine symbols that happen
; to be system primitives such as 'cons' or 'car' or 'cdr', etc.

4. Part accessors

car (1)
cdr (1)

5. Primitive predicates (all functions below can only take 1 argument)

atom?
pair?
list?
null?
integer?
real?
number? // in OurSchem, real? = number?, but not in Scheme (there are complex-numbers)
string?
boolean?
symbol?

6. Basic arithmetic, logical and string operations

+ (>= 2)
- (>= 2)
* (>= 2)
/ (>= 2)

; in evaluating 'and' or 'or', it is possible that some "argument expr"
; does not get evaluated ; use Petite Scheme to see what this means
; e.g., (set! a 5) a (and (set! a 10) #f (set! a 100)) a (or #t (set! a 200)) a

not (1)
and (>= 2)
or (>= 2)

; all functions below can take 2 or more arguments

>
>=
<
<=
=
string-append
string>?
string<?
string=?

7. Eqivalence tester

eqv? (2)
equal? (2)

8. Sequencing and functional composition

begin (>= 1)

; the user may also enter, e.g., >>(car (cdr '(1 2 3 4)))<<

9. Conditionals

; in evaluating 'if' or 'cond', it is possible that some "sub-expr"
; does not get evaluated (this is the meaning of conditional expressions) ;
; use Petite Scheme to check ;

if (2 or 3)
cond (>= 1)

10. clean-environment

; 此指令將 user 的 definitions 清空，一切重新開始

clean-environment (0)

Example : // assuming that we run the system using interactive I/O

// ========= I/O starts below and does not include this line ========
Welcome to OurScheme!

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

(cons 3 4) ; an operation on two objects
(3
.
4
)

> (cons 3
nil
) ; '(3 . nil)' = '(3)'
(3
)

> (cons 3
()
) ; same thing
(3
)

> (CONS 3 4) ; Scheme distinquishs between upper and lower cases
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

> ; 2. To "by pass" the default interpretation of an S-exp

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

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

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

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

> (cons 3 '(4321 5))
( 3
   4321
   5
)

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

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

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

> ; 2. To give a (symbolic) name to an object

; Meaning of DEFINE revisited ("令")

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

; DEFINE sets up the binding between a name and an internal data structure

abc
ERROR (unbound symbol) : abc

> (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

> (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
)
```

```
> ; Combining (1), (2) and (3)
(define hello '(1 2 . 3))
hello defined

> hello
( 1
    2
    .
    3
)

> (cons hello
              4
    )
( ( 1
        2
        .
        3
    )
    .
    4
)

> (cons hello
              '(4)
    )
( ( 1
        2
        .
        3
    )
    4
)

> (define hello "CYCU ICE (1 2 3)")
hello defined

> (cons hello
              '(400 (5000 600) 70)
    )
( "CYCU ICE (1 2 3)"
```

```
   400
   ( 5000
      600
   )
   70
)

> (define there "Number One!")
there defined

> (cons hello there)
( "CYCU ICE (1 2 3)"
   .
   "Number One!"
)

> (define hello '(1 2 . (3)))
hello defined

> (list 3 4)
( 3
   4
)

> ( list hello
               4
   )
( ( 1
      2
      3
   )
   4
)

> ; 3. Whenever a function is called, its parameters are evaluated first.
;       However, if the first symbol of a to-be-evaluated list
;       is not bound to a function in the first place, the evaluation process
;       stops, and an appropriate error message is issued.

> (f 3 b)
ERROR (unbound symbol) : f
```

```
> (cons 3 b)
ERROR (unbound symbol) : b

> (cons 3 a)
( 3
    .
    5
)

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

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

> (cons 5 a)
( 5
    3
    4
)

> a
( 3
    4
)

> ; 4. Different parts of a list (or a dotted pair) can be
;       individually accessed

(car '(3 4))        ; the "left part" of a dotted pair
3

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

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

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

> (car a)
3

> (car WarAndPeace!)
ERROR (unbound symbol) : WarAndPeace!

> (cdr '((3 4) 5)    )    ; the "right part" of a dotted pair
( 5
)

> (cdr '((3 4) "Happy New Year!" . 6)    )
( "Happy New Year!"
    .
    6
)

> (cdr '((3 4) . "Merry Christmas!")    )
"Merry Christmas!"

> (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
)

> (car 3)
ERROR (car with incorrect argument type) : 3

> (car 3 4)
ERROR (incorrect number of arguments) : car

> (car 3 . 5)
ERROR (non-list) : ( car
   3
   .
   5
)

> ; 5. Primitive predicates (A predicate is a function that returns
;           "true" or "false"; By convention, the name of a predicate
;           should have a suffix '?')

> (atom? 3)
#t

> (atom? '(1 . 2))
nil

> (pair? 3)           ; Other Lisps do not have PAIR; they have ATOM
nil

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

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

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

> (list? 3)
nil
```

```
> (list? '(1 2 3))
#t

> (list? '(1 2 . 3))
nil

> (null? ())          ; is it the empty list?
#t

> (null? #f)
#t

> (null? '(3 . 4))
nil

> (integer? 3)
#t

> (integer? +3)
#t

> (integer? 3.4)
nil

> (integer? -.4)
nil

> (real? 3)
#t

> (real? 3.4)
#t

> (real? .5)
#t

> (number? 3) ; in OurScheme, is-real IFF is-number
#t

> (number? 3.4) ; but in other Schemes, there may be complex numbers
#t
```

```
> (string? "Hi") ; therefore, in other Scheme, a number may not be real
#t

> (string? +3.4)
nil

> (boolean? #t)
#t

> (boolean? ())
#t

> (boolean? #f)
#t

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

> (symbol? 'abc)
#t

> (symbol? 3)
nil

> (number? America)
ERROR (unbound symbol) : America

> (define America '(U. S. A.))
America defined

> (symbol? America)
nil

> (pair? America)
#t

> (pair? American)
ERROR (unbound symbol) : American

> (boolean? America)
nil
```

```
> (pair? Europe 4)
ERROR (incorrect number of arguments) : pair?

> (pair? America Europe)
ERROR (incorrect number of arguments) : pair?

> (define Europe 'hi)
Europe defined

> (pair? America Europe)
ERROR (incorrect number of arguments) : pair?

> (define a . 5)
ERROR (non-list) : ( define
   a
   .
   5
)

> (define a) ; problem with the number of parameters
ERROR (DEFINE format) : ( define
   a
)

> (define a 10 20)
ERROR (DEFINE format) : ( define
   a
   10
   20
)

> (define cons 5) ; attempt to redefine a system primitive
ERROR (DEFINE format) : ( define
   cons
   5
)

>
; 6. Basic arithmetic, logical and string operations

> (+ 3 7)
```

```
10

> (+ 3 7 10 25)
45

> (- 3 7)
-4

> (- 3 7 10 25)
-39

> (/ 5 2)    ; integer division
2

> (/ 5 2.0) ; float division ; a float is always printed in 3 digits
2.500

> (/ 2 3.0) ; Use printf( "%.3f", ...) in C or String.format( "%.3f", ...) in Java
0.667

> (- 3.5 5)
-1.500

> (* 3 4)
12

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

> (* 3)
ERROR (incorrect number of arguments) : *

> (* 3 4 5)
60

> (* 1 2 3 4 5)
120

> (- 1 2 3 4 5)
-13

> (define a 5)
```

```
a defined

> (/ 15 a)
3

> (/ 7 a)
1

> (/ 15.0 3)
5.000

> (/ 30 5 0) ; always test for "division by 0" before performing division
ERROR (division by zero) : /

> (+ 15.125 4)
19.125

> (not #t)
nil

> (> 3 2)
#t

> (> 3.125 2)
#t

> (>= 3.25 2)
#t

> (< 3.125 2)
nil

> (<= 3.125 2)
nil

> (= 2 2)
#t

> (= 2 a)
nil

> (> a a)
```

```
nil

> (+ a a a)
15

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

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

> (string>? "az" "aw")
#t

> (string<? "az" "aw")
nil

> (string=? "az" "aw")
nil

> (string=? "az" (string-append "a" "z"))
#t

> (string>? "az" "aw" "ax")
nil

> (string<? "az" "aw" "ax")
nil

> (string=? "az" "aw" "ax")
nil

> (string>? "az" "aw" "atuv")
#t

> (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

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

> ; 7. 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".

; equal?, on the other hand, is the usual notion of
; equality comparison

(eqv? 3 3)
#t

> a
( 3
   4
)

> (eqv? a a)
#t

> (eqv? a '(3 4))
nil

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

> (define b a)
b defined

> (eqv? a b)
#t

> (define c '(3 4))
```

```
c defined

> (eqv? a c)
nil

> (equal? a c)
#t

> (eqv? '(3 4) '(3 4))
nil

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

> (equal? a a)
#t

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

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

> ; some functional compositions
(not (pair? 3))
#t

> (define a 5)
a defined

> ( and               ; 'and' either returns the evaluated result of
      (pair? 3)    ; the first one that is evaluated to nil
      a                  ; or the evaluated result of the last one
   )
nil

> ( and #t a )
5

> ( or                ; 'or' either returns the evaluated result of
      a                  ; the first one that is not evaluated to nil
      (null? ()) ; or the evaluated result of the last one
```

```
   )
5

> ;
   ; Let us talk about conditionals before we talk about
   ; sequencing and functional composition
   ;
   ;                    9. Conditionals
   ;

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

> (define a 5)
a defined

> (if a 'good 'bad) ; note : 'if' can take just two arguments
good

> (if #t 30)
30

> (if #f 20)
ERROR (no return value) : ( if
   nil
   20
)

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

> (define a nil)
a defined

> (if a '(1 2) '(3 4))
( 3
   4
)

> (if (not a) '((1 (2) 1) 1) '((3) (4 3)))
( ( 1
      ( 2
```

```
      )
      1
   )
   1
)

> (define b 4)
b defined

> ; 'else' is a keyword (and not a reserve word) in OurScheme
   ; (or rather, Scheme) ;
   ; according to our textbook (by Sebesta), a keyword has a
   ; special meaning ONLY WHEN it appears in some special contexts
   ; (translation: when the word appears in contexts that are not
   ;                        special, the word is just an "ordinary word")
   ; 'else' has a special meaning only when it appear as the first
   ; element of the last condition of 'cond' ;
   ; in all other cases, 'else' is considered a normal symbol

(cond ((> 3 b) 'bad)
          ((> b 3) 'good)
          (else "What happened?") ; this 'else' has a special meaning ;
)                                          ; it means "in all other cases" here
good

> (cond ((> 3 b) 'bad)
             (else 'good)                ; this 'else' is treated as a normal symbol
             (else "What happened"); this 'else' is treated as a keyword
   )
ERROR (unbound symbol) : else

> (define else #f)
else defined

> (cond ((> 3 b) 'bad)
             (else 'good)                ; the normal symbol 'else' is bound to nil
             (else "What happened"); this 'else' means "in all other cases"
   )
"What happened"

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

```
            (else "What happened?")
   )
"What happened?"

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

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

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

> (cond)
ERROR (COND format) : ( cond
)

> (cond #t 3)
ERROR (COND format) : ( cond
```

```
   #t
   3
)

> (cond (#t 3))
3

> (cond (#f 3))
ERROR (no return value) : ( cond
   ( nil
      3
   )
)

> (cond (#t (3 4)))
ERROR (attempt to apply non-function) : 3

> (cond (#f (3 4)) 5)
ERROR (COND format) : ( cond
   ( nil
      ( 3
         4
      )
   )
   5
)

> (cond (#f (3 4)) (5 6))
6

> (cond (#f (3 4)) ("Hi" (cons 5) . 6))
ERROR (COND format) : ( cond
   ( nil
      ( 3
         4
      )
   )
   ( "Hi"
      ( cons
         5
      )
      .
```

```
      6
   )
)

> (cond (#f (3 4)) ("Hi" (cons 5) 6))
ERROR (incorrect number of arguments) : cons

> (cond (#f (3 4)) ("Hi" (cons 5 6) 7))
7

> ;
   ; 8. Sequencing and functional composition
   ;
   ;       Can be more complex than what is given here

(define d 20)
d defined

> d
20

> (if #t 3 5)
3

> (begin
     3    4    5)
5

> (begin
     3 4 d)
20

> (begin
     (+ 3 5)
     (- 4 5)
     (* d d)
   )
400

> (define a 20)
a defined
```

```
> (define b 40)
b defined

> (+ d
        ( if (> a b)
                (+ a (* a b))
                (- b (+ a a))
        )
   )
20

> (+ d
        ( if (> a b)
                (+ a (* a b))
                ( begin
                   (- b (+ a a))
                   70
                )
        )
   )
90

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

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

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

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

> ;
   ; 10. clean-environment cleans up the (user-defined) environment
   ;

(clean-environment)
environment-cleaned

> a
```

```
ERROR (unbound symbol) : a

> (define a 5)
a defied

> a
5

> (clean-environment)
environment cleaned

> a
ERROR (unbound symbol) : a

> ;
   ; 11. the binding of a symbol can be a function, which is an atom too
   ;

cons ; the binding of the symbol 'cons' is a function with original name being 'cons'
#<procedure cons>

> (atom? cons)
#t

> (define myCons cons) ; let the binding of 'myCons' be the binding of 'cons'
myCons defined

> myCons ; the binding of 'myCons' is a function with original name being 'cons'
#<procedure cons>

> (define a (myCons car cdr))
a defined

> a
( #<procedure car>
   .
   #<procedure cdr>
)

> (car a)
#<procedure car>
```

```
> (cdr a)
#<procedure cdr>

> (define a (list car cdr))
a defined

> (car a)
#<procedure car>

> (cdr a)
( #<procedure cdr>
)

> ((car a) (cons car cdr)) ; just think of a function as a "value" just like 3
#<procedure car>

> (    ((car a) (cons car cdr)) ; test data like this will not appear
         '((10 20) (30 40) . 50)    ; until Prob. 6, 7, 13, 14, 15 and 16
    )
( 10
    20
)

>
(
       exit
                  )

Thanks for using OurScheme!
// ========= I/O ends above and does not include this line ========
```

// ==========

Project 2 可能會出現的 error - 總整理

一、Project 1 的四個可能會出現的 error，在 Project 2 依舊可能會出現：

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 >>...<<
ERROR (no closing quote) : END-OF-LINE encountered at Line X Column Y
ERROR (no more input) : END-OF-FILE encountered

二、Project 2 增加了以下的這些 error：

```
> (cons 3 . 5)
ERROR (non-list) : ( cons
   3
   .
   5
)

> (cons 3 (cons 3 . 5))
ERROR (non-list) : ( cons
   3
   .
   5
)

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

> (exit 0)
ERROR (incorrect number of arguments) : exit

> (car 3)
ERROR (car with incorrect argument type) : 3

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

> (if #f 3)
ERROR (no return value) : ( if
   nil
   3
)
```

```
> (cond (#f 3) (#f 4))
ERROR (no return value) : ( cond
   ( nil
      3
   )
   ( nil
      4
   )
)

> noSuchThing
ERROR (unbound symbol) : noSuchThing

> (cons noSuchThing noOtherThingEither)
ERROR (unbound symbol) : noSuchThing

> (/ 30 5 0)
ERROR (division by zero) : /

> (define a) ; problem with the number of parameters
ERROR (DEFINE format) : ( define
   a
)

> (define a 10 20)
ERROR (DEFINE format) : ( define
   a
   10
   20
)

> (define cons 5) ; attempt to redefine a system primitive
ERROR (DEFINE format) : ( define
   cons
   5
)

> ;
   ; >>ERROR (COND format)<< will only be tested in Problems 13, 14
   ; and 16
   ;
```

```
(clean-environment)
environment cleaned

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

> (cond)
ERROR (COND format) : ( cond
)

> (cond #t 3)
ERROR (COND format) : ( cond
   #t
   3
)

> (cond (#f 3))
ERROR (no return value) : ( cond
   ( nil
      3
   )
)

> (cond (#t (3 4)))
ERROR (attempt to apply non-function) : 3

> (cond (#f (3 4)) 5)
ERROR (COND format) : ( cond
   ( nil
      ( 3
         4
      )
   )
   5
)

> (cond (#f (3 4)) ("Hi" (cons 5) . 6))
ERROR (COND format) : ( cond
   ( nil
```

```
        ( 3
            4
        )
    )
    ( "Hi"
        ( cons
            5
        )
        .
        6
    )
)

> (cond (#f (3 4)) ("Hi" (cons 5) 6))
ERROR (incorrect number of arguments) : cons
```