2021 spring, PL project 3 (OurScheme Project 3)

Due : 6/27, 2021(Sunday) before midnight

==================================================================

For this project, you need to extend EvalSExp(), so that it is capable of

evaluating user-defined functions.

In order to do so, you must first extend your implementation of DEFINE, so

that the user can define a function before he/she calls such a function.

You also need to allow the creation and use of "local definitions" via the

use of the 'let' construct.

The use of "nameless functions" (via the use 'lambda') should also be allowed.

In other words, the main focus of Proj. 3 consists of three special "forms":

'let', 'lambda', and 'define'.

In addition, you must also handle error cases (see Part II below).

==================================================================

Part I - Basic requirement

==================================================================

在正式介紹Project 3之前，我們必須先釐清一個概念：

誰是function？ 誰不是function？ 如果不是function，那是什麼？

There are ten reserve words in OurScheme. Below is a list of these

reserve words : // according to our textbook, a 'reserve word' is a

// word that is reserved for the system to use

quote

and

or

begin

if

cond

define

lambda

set!

let

'let', 'lambda' and 'define' are three of the above mentioned reserve

words. They are not functions. Whenever a reserve word appears, the

system should check the syntax of the related code fragment.

Though S-expressions starting with any one of the above ten reserve

words are actually "forms" and not functions, some of them may

nevertheless return values. For this reason, we will also refer to

these "forms" as "functional forms".

本學期的OurScheme project不會作類似以下要求(但將來的OurScheme project會)

> define // or 'quote' or 'begin' or ... (總共十個cases)

DEFINE format

> (define abc quote) // or 'define' or 'begin' or ...

QUOTE format

==================================================================

※ let

The syntax of 'let' is the following :

( let ( ... ) ......... )

where

(a) '...' is a sequence of S-expressions, with each S-expression being

of the form

( SYMBOL S-expression )

(b) '.........' is a non-empty (!!!) sequence of S-expressions.

In words, 'let' has at least two parameters.

Its first argument is a list of zero or more pairs, where each pair

must be of the form : ( SYMBOL S-expression)

The working of 'let' is as follows :

＊ The '...' part defines local symbols with bindings.

e.g.,

if '( ... )' is

( ( x 5 )

( y '(1 2 3))

)

then

two local symbols 'x' and 'y' are defined

AND

'x' is bound to the atom 5, while 'y' is bound to the list (1 2 3).

＊ The '.........' are normal S-expressions. These S-expressions

are such that

(i) The "LET-defined" local variables (i.e., 'x' and 'y') can appear

in these S-expressions, and the system knows what their bindings

are.

(ii) The evaluated result of the last S-expression in '.........'

is taken to be the evaluated result of the entire LET expression.

Example :

> (clean-environment)

environment cleaned

> ( let ( (x 3) (y '(1 2 3))

)

(cons 1 '(4 5)) ; this will be evaluated ; but no use

(cons x (cdr y)) ; the value of this one is the value of LET

)

( 3

2

3

)

> x

ERROR (unbound symbol) : x

If there is anything syntactically wrong with the syntax of 'let',

the system should print : ERROR (let format)

Example :

> (let (car '(1 2 3)) ; first argument of 'let' should be a list of pairs

; moreover, there ought to be a second argument

)

ERROR (let format)

> (let ((x 3 4)) 5 ; first argument of LET should be a list of

; pairs ; '(x 3 4)' is not an acceptable pair

)

ERROR (let format)

> (let ((x 3)

)

5

)

5

> (let ( ( (car '(x y z)) ; first argument of LET should be a list of pairs

3

) ; Furthermore, the first element of each

) ; pair must be a symbol

5

)

ERROR (let format)

> (let () ; There should be at least one S-expression following

; the first argument

)

ERROR (let format)

> (let () 5

)

5

> (let ( ( ( car '(x y z))

5

)

)

)

ERROR (let format)

> (let ( ( x (cons 5) ) ; the problem is not in LET-format

)

( + x x )

)

ERROR (incorrect number of arguments) : cons

> (let ( ( x (cons 5) )

)

)

ERROR (let format)

> (let ((x (1 2 3))) 5) ; LET-format OK

ERROR (attempt to apply non-function) : 1

> (let ((x (1 2 3))

)

)

ERROR (let format)

※ lambda

The syntax of 'lambda' is :

( lambda ( zero-or-more-symbols ) one-or-more-S-expressions )

A lambda expression defines a (nameless) function. The evaluation of

this lambda expression returns the function it defines.

A lambda expression has two or more parameters.

The first argument is a list of zero-or-more-symbols (these symbols

are the arguments of the function being defined by the lambda expression).

The one-or-more-S-expressions constitute the function's body.

Example :

> (clean-enviornment)

environment cleaned

> ( lambda )

ERROR (lambda format)

> ( lambda x )

ERROR (lambda format)

> ( lambda x y z )

ERROR (lambda format)

> ( lambda (x) y z ; the evaluation of a lambda expression

; produces an internal representation of a

) ; function

#<procedure lambda>

> ( lambda (x) )

ERROR (lambda format)

> ( lambda () y ) ; this function just returns the binding of 'y'

#<procedure lambda>

> ( lambda (5) y )

ERROR (lambda format)

> ( lambda () 5 )

#<procedure lambda>

> ( lambda () () )

#<procedure lambda>

> ( lambda () )

ERROR (lambda format)

> ( lambda () (+ c 5)

)

#<procedure lambda>

> ( ( lambda () (+ c 5) ; first, the internal representation of a function

) ; is produced ; this internal representation

; is "the evaluated result of the first argument"

; once the binding of the first argument (of

; the top-level list) is obtained and found

; to be a function, that function is applied ;

)

ERROR (unbound symbol) : c

> ( ( lambda () (+ 5 5) (+ 5 6)

)

)

11

> ( ( lambda () (+ 5 5) (+ c 6)

)

8

)

ERROR (incorrect number of arguments) : lambda expression

※ define

The syntax of 'define' is :

( define SYMBOL S-expression )

OR

( define ( SYMBOL zero-or-more-symbols ) one-or-more-S-expressions )

Moreover, a DEFINE-expression must appear at the top-level (i.e., it

cannot be an "inner" expression).

The first define-expression defines the binding of a symbol. We have

seen how this define-expression can be used in the previous projects.

With a proper use of the lambda-expressions, we can also define functions

using the first define-expression.

The second define-expression is only used for defining functions.

Example :

> (clean-environment)

environment cleaned

> ( define a 2 )

a defined

> ( define f ( lambda (x) (+ x x c) ; the binding of 'f' is defined

) ; to be the internal representation

) ; of a function

f defined

> f

#<procedure lambda>

> (f 1 2 3)

ERROR (incorrect number of arguments) : lambda

> (f a)

ERROR (unbound symbol) : c

> (f b)

ERROR (unbound symbol) : b

> ( define c 10 )

c defined

> (f a)

14

> ( define ( g x ) (h x x) )

g defined

> g

#<procedure g>

> (g 3)

ERROR (unbound symbol) : h

> ( define ( k x ) (h z z) )

k defined

> (k w)

ERROR (unbound symbol) : w

> (k c)

ERROR (unbound symbol) : h

> (define (h x y) (+ x 20 a))

h defined

> (g c)

32

> ( define (h x y) )

ERROR (define format)

> ( define x 10 20 )

ERROR (define format)

> ( define x 300 ) ; 'x' is a "global"

x defined

> (g c) ; global x != parameter x

32

> (define cadr (lambda (x) (car (cdr x))))

cadr defined

> cadr

#<procedure lambda>

> (cadr '(1 2 3 4))

2

> (define (cadr x) ( (lambda (x) (car (cdr x)))

x

)

)

cadr defined

> (cadr '(1 2 3 4))

2

> cadr

#<procedure lambda>

> cons

#<procedure cons>

==================================================================

Part II - Error handling (the "no return value" error)

==================================================================

現在介紹Proj. 3 (與Proj. 4)的一個重要run-time error: no return-value.

說來話長，請耐心看完。

1. total function vs. partial function

我們所熟悉的functions都是total functions，亦即：

只要所有的parameters皆為此function可接受的parameters

(如Factorial的一百零一個參數是個大於或等於0的整數)、

此function就(保證、或曰應該)會return一個值。

舉例而言，C/Java的non-void functions就是(理論上應)如此。

但functions事實上還有partial functions，亦即：

雖然所有的parameters皆為此function可接受的parameters，

但此function未見得保證return一個值。

(換言之，雖然所有的parameters皆為此function可接受的parameters，

但有可能此function並不return一個值。

)

例：

(define (F x) (cond ((> x 5) x))) ; So, (F 3) has no return value

也請注意：以上(與以下)有關「function」的說明也適用於「functional form」.

2. In OurScheme, we are dealing with partial functions and not

total functions.

3. 有時，it is acceptable that a function call does not return a value.

例：

(begin (F 3) 5) ; 假設the function F已定義於上

; 雖然(F 3) does not return a value, but it is OK.

; 因為(F 3)是否return a value並不重要

(begin (begin (F 3)) 5) ; 中間的begin並未return a value. It is OK too.

4. 那...

Under what circumstances is it an error not to return a value？？？

茲將這些circumstances條列於下：

(a) When a function is called, all its actual parameters must evaluate

to a binding.

說明：此處的重點是「when a function ia called」

例：

(cond ((> 5 3) 15)

(#t (cons (F 3) (F 3)))

)

的執行不會有error，因為(cons (F 3) (F 3))不會被呼叫。

但

(cons (F 3) 5)

的執行就有error了，因為cons有被呼叫、而(F 3)無return value。

(b) When an IF or COND is evaluated and a test-condition of this IF or

COND gets evaluated, the evaluation of this test condition must

result in a binding.

說明：IF也就罷了(IF只有一個test condition，也一定會要evaluate這個

test condition (除非IF本身沒有被evaluate))

COND可能有好幾個test conditions，在evaluate這個COND的過程之中，

並非所有的test conditions都會被evaluate，例子：

(cond ((> 5 3) 15)

((F 3) (cons 4 5))

) ; the evaluation of this COND will be OK (no error!)

(cond ((< 5 3) 15)

((F 3) (cons 4 5))

) ; the evaluation of this COND is not OK ((F 3) has no return value)

所以重點是「When IF/COND is evaluated and a test-condition of this

IF/COND gets evaluated」，只有在此時、the evaluation of 此

test condition才一定必須要有return value。

(c) When an AND or OR is evaluated and a condition of this AND or

OR gets evaluated, the evaluation of this condition must

result in a binding.

說明： We are talking about

( and <condition-1> <condition-2> ... <condition-N> )

or

( or <condition-1> <condition-2> ... <condition-N> )

有error或無error的道理與(b)類似

(d) When DEFINE or SET! or LET is evaluated, the "to be assigned"

must evaluate to a binding.

說明： We are talking about

( define <symbol> HERE )

or

( set! <symbol> HERE ) ; set!將於Proj. 4出現

or

( let ( ( <symbol-1> HERE )

( <symbol-2> HERE )

...

( <symbol-N> HERE )

)

...

)

When this DEFINE or SET! or LET is evaluated,

what appears at HERE must evaluate to a binding.

Otherwise, there is no way we can initialize the

corresponding symbol.

(e) When a function or functional form is evaluated at the top level,

it must evaluate to a binding.

說明： 當OurScheme的使用者於the prompt level(亦即當system print

'> '之後)輸入一個S-expression叫system去evaluate時，

the user expects to see a return value (which is what the

system prints as a response of user input).

因此，如果此時the evaluation of the (user-input) S-expression

does not result in a binding，the system should print

>>ERROR (no return value) : ...<<

5. The error message to show when a return value is required but no values returned

(a) If a function is called and some of its actual parameters does not

evaluate to a binding, then the first occurrence of such cases

should lead to the following error message (and the control goes

back to the top level):

>>ERROR (unbound parameter) : <code of the actual parameter><<

(b) If an IF or COND is evaluated and the evaluation of a test condition

does not result in a binding, then the following error message should be

printed (and the control goes back to the top level):

>>ERROR (unbound test-condition) : <code of the test-condition><<

(c) If an AND or OR is evaluated and a condition of this AND or

OR does not evaluate to a binding, then the following error message should be

printed (and the control goes back to the top level):

>>ERROR (unbound condition) : <code of the condition><<

(d) If DEFINE or SET! or LET is evaluated and the "to be assigned"

does not evaluate to a binding, then the following error message should be

printed (and the control goes back to the top level):

>>ERROR (no return value) : <code of the "to be assigned"><<

例： > (define a (F 3)) ; F is what has been defined in the above

ERROR (no return value) : ( F

3

)

> (let ( (a 5)

(b (F 3))

)

(\* a b)

)

ERROR (no return value) : ( F

3

)

(e) If a function or functional form is evaluated at the top level and

it does not evaluate to a binding, then the following error message should be

printed (and the control goes back to the top level):

>>ERROR (no return value) : <code entered at the top level><<

(f) If an S-expression of the form >>( ( 。。。 ) ... )<< is evaluated and

the evaluation of >>( 。。。 )<< does not result in a binding (i.e.,

the evaluation of >>( 。。。 )<< results in a null pointer), then the following

error message should be printed (and the control goes back to the top level):

>>ERROR (no return value) : <pretty print form of ( 。。。 )>

Note, however, that there are two exceptions to (e): DEFINE and CLEAN-ENVIRONMENT.

When DEFINE or CLEAN-ENVIRONMENT is evaluated, it does not return any binding.

Q: How did the interaction below happen?

> (define a 5)

a defined

> (clean-environment)

environment cleaned

A: It is the system primitive reponsible for handling'(define a 5)' (or '(clean-environment)')

that prints >>a defined<< (or >>environment cleaned<<) (note that there is also a line-enter).

In other words, the main evaluation loop does not print any feedback message for DEFINE

and CLEAN-ENVIRONMENT, unless there are errors.

Q: What is the return value of 'exit'?

A: It does not matter what (a call to) 'exit' returns, because the system will not have

the chance to print what 'exit' returns.

Please consult HowToWriteOurScheme.doc to see "when to print what" when there

may be multiple errors in the being evaluated code.

==================================================================

Part III - Two extra functions for coping with

the printing of information for

DEFINE and CLEAN-ENVIRONMENT

==================================================================

There are two more functions you need to implement:

\* (verbose nil) vs. (verbose #t) ; #t can be replaced by any S-expression

; that evaluates to NOT NIL

\* (verbose?)

例： ; the "verbose" mode controls whether the system will

; print something when the being evaluated S-expression

; is DEFINE or CLEAN-ENVIRONMENT

> (verbose?) ; Is the verbose mode ON?

#t

> (define a 5)

a defined

> (clean-environment)

environment cleaned

> (verbose nil) ; let us turn off the verbose mode

nil

> (verbose?)

nil

> (define a 5)

> (clean-environment)

> (verbose 5) ; let us turn the verbose mode back on

#t

> (verbose?)

#t

> (define a 5)

a defined

> (clean-environment)

environment cleaned

>

The reason for having 'verbose' (and 'verbose?') will become clearer

in Proj. 4, when 'eval' comes into play.

==================================================================

Part IV - Proj. 3 題目的設計

==================================================================

新的(2017)「Proj. 3」的題目安排如下

(1)~(5)無error，第五題的隱藏數據是前四題的隱藏數據的"加總"

(6)~(10)有error，第十題的隱藏數據是前四題的隱藏數據的"加總"

除了「from simple to complex」之外，

有關cond, if, lambda, and, or, let的測試數據的安排，是依照以下原則

(1) define + lambda (用para.做為(initialized)"local para") - basic - incl.: COND IF BEGIN AND OR

(2) define + lambda (用para.做為(initialized)"local para") - complex - COND IF BEGIN AND OR (nested calls)

(3) (2) + functional composition // functions 呼叫 functions

(4) (3) + let (local vs. global)

(5) (4) + nested locals vs. globals + (1)~(4)集大成

(6) (1) + error tests

(7) (2) + error tests

(8) (3) + error tests

(9) (4) + error tests

(10) (5) + error tests + (6)~(9)集大成

(11) (5) + Proj. 2 集大成test(s) (no error cases)

(12) (10) + Proj. 2 集大成tests(s) (error cases)