CSCI3136
Assignment 8

Instructor: Alex Brodsky

Due: 3:00pm, Monday, March 24, 2014

In this assignment we will further expand the Scheme Evaluator to handle closures. We will first need to add two productions to the grammar from the previous assignment:

$$\begin{aligned} \textit{Proc} &\rightarrow \texttt{set!} \\ \textit{Proc} &\rightarrow \texttt{lambda} \end{aligned}$$

That is, we have added two more special functions (`set!` and `lambda`) to our evaluator.

The `set!` keyword is used to modify variables in the current scope. The syntax for `set!` is

(`set!` id atom)

where the *id* denotes a bound identifier that is in scope and *atom* is the expression to be evaluated whose result is then bound to the *id*. The result of the `set!` expression is an empty string `""`, just like for `define`. For example, the expression below yields 13.

```
( let ( ( x 42 ) )
      ( set! x 13 )
      x
)
```

The `lambda` keyword allows us to create closures (functions with a reference environment) that can then be invoked. I.e., just like you can define and use methods in Java or functions in C, we can define and use closures in Scheme. The syntax for `lambda` is

(`lambda` (param1 ...) atom ...)

where the first argument to `lambda` is a list of parameters, which are identifiers, and the remaining arguments are atoms that are evaluated *only* when the closure is invoked. Typically, `lambda` expressions are used in conjunction with `define` or `let` expressions. In Example 1, the expressions:

Example 1

```
( define area ( lambda ( l w )
                ( * l w ) ) )
( define square ( lambda ( w )
                  ( area w w ) ) )
( area 51 42 )
( square 7 )
( define execute ( lambda ( c a )
                 ( c a a ) ) )
( execute area 51 )
```

Example 2

```
( let ( ( x 42 ) )
   ( let ( ( foo ( lambda ( a )
                   ( / x a ) ) ) )
         ( let ( ( x 21 ) )
               ( foo 3 ) )
         ( set! x 14 )
         ( foo 7 ) ) )
```

first defines a name *area* and binds it a closure to compute the area given the length (l) and width (w). The second expression defines another name *square* and binds it to a closure that computes the square area, given a width, by invoking the *area* closure. The next two expressions simply invoke the *area* and *square* closures. The fifth expression binds the name *execute* to a closure that takes two parameters: a closure (c) and an atom (a), and evaluates the closure, passing the argument a to it twice. The last expression is an invocation of the closure bound to *execute*. The evaluation of these expressions is: `2142 49 2601`.

In Example 2, the expression comprises a `let`, which binds x to 42 and then evaluates a second `let` expression, which binds *foo* to a closure that divides x by the passed argument (a), then evaluates three expression: a third `let` expression, which binds binds x to 21 and then invokes *foo* with argument 3; a `set!` expression, which changes the value of x to 14; and another invocation of `foo` with argument 7. The evaluation of this expressions is 2.

An evaluator, called `fuscm.py` (written in Python) that parses and evaluates programs using the above grammar is provided as part of this assignment. This is a solution to Assignment 7. However, this evaluator does not implement `set!` or `lambda`, but it accepts expressions containing these keywords.

1. **[30 marks]** Using the provided evaluator `fuscm.py`, or your own version of an evaluator (implemented in a language of your choice) implement the `set!` and `lambda` operations. Note: the provided implementation will parse `set!` and `lambda` expressions, but does not evaluate them. So, all you need to do is modify the evaluation phase of the evaluator (unless you are creating your own).

   Note that Scheme uses lexical scoping and deep binding so your implementation must also implement the same semantics when creating and passing closures as arguments.

   A test solution will be provided for you to test your new evaluator.

   Since the choice of language is up to you, you must provide a standard script called `runme.sh` to run your parser. See previous assignment for an example.

   To submit this part of the assignment please use SVN as well as in hard-copy. On the hard-copy please note the login id of the person who submitted the programming portion of the assignment for the group. Please see the Resource or Assignment page of the course website for how to use SVN. Remember, you need to include a script called `runme.sh` that will let the marker run your code.

2. Your implementation in Question 1 should implement lexical scoping and deep binding.

   (a) **[5 marks]** What changes would you need to make to your evaluator to implement dynamic scoping?

   (b) **[5 marks]** What changes would you need to make to your evaluator to implement shallow binding?

3. **[10 marks]** Suppose you wanted to determine whether a given Scheme interpreter used lexical or dynamic scoping and shallow or deep binding. Come up with a Scheme expression that evaluates to one of four results: `(lexical shallow)`, `(lexical deep)`, `(dynamic shallow)`, or `(dynamic deep)` depending on which of the four possible combinations that the interpreter implements.

CSCI3136: Assignment 8

Winter 2014

Student Name	Login ID	Student Number	Student Signature

	Mark	
Question 1		/30
Functionality	/15	
Structure	/15	
Question 2		/10
Question 3		/10
Total		**/50**

Comments:

Assignments are due by 3:00pm on the due date before class and must include this cover page. Assignment *must* be submitted into the assignment boxes on the second floor of the Goldberg CS Building (by the elevators).

Plagiarism in assignment answers will not be tolerated. By submitting their answers to this assignment, the authors named above declare that its content is their original work and that they did not use any sources for its preparation other than the class notes, the textbook, and ones explicitly acknowledged in the answers. Any suspected act of plagiarism will be reported to the Facultys Academic Integrity Officer and possibly to the Senate Discipline Committee. The penalty for academic dishonesty may range from failing the course to expulsion from the university, in accordance with Dalhousie Universitys regulations regarding academic integrity.