Assignment #2

ECEN 5023, CSCI 7135

Due February 5, 2008

Implement four interpreters for the lambda calculus in Objective Caml. You can choose to implement call-by-value, call-by-name, or any other evaluation strategy.

Recall the grammar for the lambda calculus:

```
e ::=                   expressions
      x                   variables
    | lambda x. e         function creation
    | e e                 function application

v ::=                   values
      lambda x. a         function creation
```

You can find some example parsers and interpreters on the web site for the course textbook. You have learned several ways to deal with variables:

1. Variables are symbols, e.g., `x`, `y`, and `z`.
2. Variables are DeBruijn indices.
3. Locally nameless: bound variables are DeBruijn indices and free variables are symbols.

You have also learned several ways to handle function application and the replacement of parameters by arguments.

1. Substitution.

   ```
   (lambda x. e1) e2 --> [x->e2] e1
   ```

2. Environment passing. In the folowing big-step semantics, the environment `G` is a function from variables to expressions (or values for pass-by-value). Environments can be implemented using hash tables, association lists (linked lists of key-value pairs), or even functions.

   ```
   ----------------------------------------
   G |- (lambda x. e) => G |- <lambda x.e, G>
   ```

```
--------------
G |- x => G(x)


G |- e1 => <lambda x.e3, G'>    G |- e2 => e4     G'(e->e4) |- e3 => e5
-------------------------------------------------------------------------
G |- e1 e2 => e5
```

Finally, you have learned several styles of evaluation.

1. Small-step semantics with
   (a) congruence rules:

```
      e1 |--> e1'
   ------------------
    e1 e2 |--> e1' e2

      e2 |--> e2'
   ------------------
    v1 e2 |--> v1 e2'
```

   (b) evaluations contexts:

```
   E ::= [] | E e | v E

      e --> e'
   --------------------
    E[e] |--> E[e']
```

2. Big-step semantics. Here's an example of big-step semantics using substitution.

```
--------------------------
lambda x. e => lambda x. e


e1 => lambda x.e3     e2 => e4      [x->e4]e3 => e5
-----------------------------------------------
              e1 e2 => e5
```

3. Abstract machines:
   (a) that use a procedure stack (like the SECD).

```
compile to bytecode
[|x|] = x
```

```
[|(e1 e2)|] = [|e1|] [|e2|] app
[|lambda x.e|] = fun x. [|e|]

S - stack of values
E - environment, maps variables to values
C - control
D - dump (the procedure call stack)


variable lookup
(S,E,x#C,D)                    |--> (E(x) S, E, C, D)

closure creation
(S,E, (fun x.C')#C, D>      |--> (<fun x.C',E> S, E, C, D)

function application
(v#<fun x.C',E'>#S, E, app#C, D) |--> ([], E'(x<-v), C',(S,E,C,D))

function return
(v#S, E, [], (S',E',C',D))   |--> (v#S', E', C', D)
```

(b) that use an continuation stack (like the CEK machine). (This machine works directly on programs. No compilation to bytecode is needed.)

```
c ::= <e,E>    (closure, expression and environment)
k ::= [] | arg c k | fun v k     continuations

variable lookup
(<x,E>, k)                     |--> (E(x),k)

application expression: evaluate the function expression
(<(e1 e2), E>, k)              |--> (<e1,E>, arg <e2,E> k)

function call, evaluate body
(<v,E>, fun <lambda x.e,E'> k) |--> (<e,E'(x<-<v,E>)>, k)

application expression: evaluate the argument expression
(<v,E>, arg <e,E'> k)          |--> (<e,E'>, fun v E k)
```

For each of your four implementations, choose one option from each of the above selections and implement the approach. You may not choose a selection that is identical to the interpreters on the textbook web site. Make sure to include many example programs and check that your interpreter gives the expected output for those examples.