Machine Problem Set 4

Assigned: March 3, 2006
Due: March 28, 2006

Objectives and Background

This MP will introduce you to interpreters, and further your understanding of environments. You may wish to read chapter 3 of Essentials of Programming Languages before starting this MP.

Problem 1  Lexical Addressing (10 points)

Write a lexical address translator, called lex-addr-trans. It should replace each instance of var-exp with an appropriate instance of lexvar-exp. A description of lexical addresses can be found in section 1.3.2 of EoPL. The basic idea is to remove any need for variable names. Variables are specified by how many environments up from the current environment the variable is (lexical depth) and their position within the environment.

For example, something like:

\[
\text{let } a = 0 \text{ in } \\
\text{let } f = \text{proc}(b, c) \text{ +(+(a, b), c) } \\
\text{in (f 1 2)}
\]

should be turned into:

\[
\text{let } a = 0 \text{ in } \\
\text{let } f = \text{proc}(b, c) \text{ +(+(: 1 0, : 0 0), : 0 1) } \\
\text{in (: 0 0 1 2)}
\]

...except that lex-addr-trans should work on abstract syntax trees. I.e.,

\[
> \text{(equal? (lex-addr-trans (scan&parse <example1>)) (scan&parse <example2>))} \\
\#t
\]

Also, implement a case statement for lexvar-exp in eval-expression such that programs translated with lex-addr-trans, or that use lexvar-exp work correctly. I.e.,
> (eval-program (lex-addr-trans (scan&parse <program>)))

> (run <program>)

...should both return the same results for all programs.
Note: lex-addr-trans does not have to handle proc2, dynproc, letrec, and letref, as you will not have implemented the functionality for those expressions yet.

Problem 2  Dynamic Scoping (10 points)

Implement the dynproc expression. This should create a closure that is dynamically scoped. You will probably also need to modify apply-procval, so that dynamically scoped closures know what environment they are dealing with.

> (run "let x = 0 in
    let f = dynproc (y) +(x, y) in
    let g = proc (x) (f 42) in
    (g 2778)"")

2820

If f were statically scoped, the x would refer to the x declared in let x = 0. However, since f is dynamically scoped, it refers to the x in it’s calling environment - the argument passed to g.

Note: Do not worry about the interaction of dynproc and lexvar-exp. (Although you may want to think about why they would interact oddly.)

Problem 3  Environment and Store (20 points)

For this problem, you will be modifying how the environments are handled by the interpreter. Reviewing problems 4 - 6 in MP2 may be helpful here. You will want to implement a store (variable length array), and have the environment store indexes to locations within the store. The new environment should consist of a store that stores the values of various variables in the environment, and an environment that stores pairs of names and indices, indicating where in the store the value of each variable is stored.

If done correctly, the changes should make no real difference in how the interpreter works. You may wish to verify that things are working correctly by occasionally printing out the contents of the store.

> (run "let x = 0 in
    let f = proc (y) +(x, y) in
    (f 2778)"")

< 0 >
Your store should have the following functions as an interface: empty-store, get-store, set-store, get-empties.

Note: Make sure that these are the only functions that directly touch the contents of the store - i.e., you should be able to swap in any other store implementation that uses the same interface, and your code should still work.

> (define store (empty-store))
> (get-empties store 5)
<list of the first 5 free indexes>
> (define store (set-store store <1st index> 'abc))
> (get-store store <1st index>)
abc
> (define store (set-store store <2nd index> 123))
> (get-store store <1st index>)
abc
> (get-store store <2nd index>)
123

Problem 4  Letrec, Letref, and Set (20 points)

Write letrec, letref, and set.

letrec is the recursive let. As described in EoPL, there are several ways to implement letrec. letrec can be defined using Scheme’s letrec (which only works if the underlying language allows recursion), by creating the environment and then creating the closures when needed, or by creating the environment and then filling in the values afterwards. You will be using the third method. Your implementation should be similar to the implementation in figure 3.12. However, since values are stored in the store, you can extend the environment, use the new environment to evaluate the right hand sides, and then store the values in the store.

letref is the call-by-reference version of let. If the right hand side is a variable, you should make the new variable be a reference to the old variable. Thus if the either variable has a new value assigned to it, both variables will reflect the new value.

set is used to assign values to variables. It should return 1 - not that I expect to test it, but in case you were curious.

> (run "let x = 0 in
       letref y = x in
       let d = set y = 42 in
       x")
Problem 5  Calling Conventions (10 points)

This problem is only required for all the graduate students. Undergraduates may do this problem for extra credit.

Implement proc2, a procedure definition that allows for three different types of calling conventions. Each variable in the argument list for the procedure will be proceeded by either val, ref, need. The keywords refer to call-by-value, call-by-reference, and call-by-need, respectively.

> (run "let z = 2778
    inc = proc2(ref x) let d = set x = +(x, 1) in x
    in begin (inc z) ; z end")
2779

> (run "let z = 42
    f = proc2(ned x) 1
    g = proc2(ref x) let d = set x = 2778 in x
    in begin (f (g z)) ; z end")
42

> (run "let z = 42
    f = proc2(ned x) x
    g = proc2(ref x) let d = set x = 2778 in x
    in begin (f (g z)) ; z end")
2778

Note: Call-by-need will most likely require you to modify the store when retrieve a value that was passed into a function via call-by-need. Unfortunately, the way get-store is specified, there is no way to return a modified store. Thus, it will be acceptable to modify the store passed in, in a get-store call, if you are dealing with a value passed via call-by-need (i.e., get-store can act in a destructive manner in this instance). (After further consideration, it occurs to me that this is not strictly necessary - however it does result in a cleaner set of code, so feel free to use the above.)
Handin

You should hand in a single file named `mp4.scm’ with the implementations of the above functions. The names of the functions and number and order of arguments of the functions should be the same as in the problems.

Please see the CS 421 FAQ web page for handin instructions.