Homework 4

Jason Hemann

December 19, 2017

1 Preamble

- Do consult class notes, online lecture notes, and test cases when completing this assignment.
- Zip your `main.ss` and `pmatch`, using a command like the following: (`zip <username>-hw4.zip main.ss pmatch.ss`)
- Load `pmatch.ss` in the top of your file: `(load "pmatch.ss")`
- You should define several functions.
- You should be writing recursive functions over our extended language of `lambda`-calculus expressions.
- Upload the `.zip` file to the PLC grading server
- Your interpreters must work for at least these test cases. Of course, these tests are not exhaustive; you should use your own tests as well.

2 Part 1 : Revenge of the `lex`

When we implemented lex before, it could handle variables, application, and `lambda`-abstraction forms. Extend your previous definition of `lex` so that it can handle not only those forms, but also numbers, `zero?`, `sub1`, `*`, `if`, and `let`. This should be a straightforward extension (`let` should be the only line that requires any real effort), but it also serves as a chance to improve a misbehaving lex from an earlier assignment. In order to disambiguate numbers from lexical addresses, you should transform a number `n` into `(const n)`.

```
> (lex '((lambda (x) x) 5)  '())
((lambda (var 0)) (const 5))
> (lex '(lambda (!)
          (lambda (n)
            (if (zero? n) 1 (* n (! (sub1 n))))))
        '())
(lambda
  (lambda
    (if (zero? (var 0))
        (const 1)
```

```
        (* (var 0) ((var 1) (sub1 (var 0)))))))
> (lex '(let ((! (lambda (!)
                   (lambda (n)
                     (if (zero? n) 1 (* n ((! !) (sub1 n))))))))
          ((! !) 5))
       '())
(let (lambda
       (lambda
         (if (zero? (var 0))
             (const 1)
             (* (var 0) (((var 1) (var 1)) (sub1 (var 0)))))))
  (((var 0) (var 0)) (const 5)))
```

3 Part 2 : Dynamic Scope

The second part of this week's assignment is to create an interpreter that uses dynamic scope.

3.1 Explanation/recapitulation of dynamic scope

So far, we have implemented our interpreters so that, if there are variables that occur free in an a procedure, they take their values from the environment in which the lambda expression is defined. We accomplish this by creating a closure for each procedure we see, and we save the environment in the closure. We call this technique static binding of variables, or static scope. Lexical scope is a kind of static scope.

Alternatively, we could implement our interpreters such that any variables that occur free in the body of a procedure get their values from the environment from which the procedure is called, rather than from the environment in which the procedure is defined.

For example, consider what would happen if we were to evaluate the following expression in an interpreter that used lexical scope:

```
(let ([x 2])
  (let ([f (lambda (e) x)])
    (let ([x 5])
      (f 0))))
```

Our lexical interpreter would add `x` to the environment with a value of `2`. For `f`, it would create a closure that contained the binding of `x` to `2`, and it would add f to the environment with that closure as its value. Finally, the inner let would add `x` to the environment with a value of `5`. Then the call `(f 0)` would be evaluated, but since it would use the value of `x` that was saved in the closure (which was `2`) rather than the value of `x` that was current at the time `f` was called (which was `5`), the entire expression would evaluate to `2`.

Under dynamic scope, we wouldn't save the value of `x` in the closure for `f`. Instead, the application `(f 0)` would use the value of `x` that was current in the environment at the time it was called, so the entire expression would evaluate to `5`.

3.2 Your task

Define `value-of-dynamic`, an interpreter that implements dynamic scope. You can start with the dynamically-scoped interpreter we wrote in class that used `let` and `pmatch`. You should be able to share use your environment helpers from a previous assignment, but you should not implement an abstraction for closures in this interpreter. Instead, the value of a lambda abstraction should be that same lambda abstraction. In the same way the value of a number is that same number. You'll find then, that when you go to evaluate an application, there's only one environment in which you can evaluate the body. This is a pretty simple change. To liven things up a little (and also to allow us a more interesting test case), this interpreter should also implement `let`, `if`, `*`, `sub1`, `null?`, `zero?`, `cons`, `car`, `cdr`, and `quote`. When evaluating the expression `(cons 1 (cons 2 '()))` `value-of-dynamic` should return `(1 2)`. Now `quote` is a bit of a tricky beast. So here's the quote line for the interpreter.

```
[`(quote ,v) v]
```

```
> (value-of-dynamic
    '(let ([x 2])
       (let ([f (lambda (e) x)])
         (let ([x 5])
           (f 0))))
    (empty-env))
5
> (value-of-dynamic
    '(let ([! (lambda (n)
                (if (zero? n)
                    1
                    (* n (! (sub1 n)))))])
       (! 5))
    (empty-env))
120
> (value-of-dynamic
    '((lambda (!) (! 5))
        (lambda (n)
          (if (zero? n)
              1
              (* n (! (sub1 n))))))
    (empty-env))
120
> (value-of-dynamic
    '(let ([f (lambda (x) (cons x l))])
       (let ([cmap
              (lambda (f)
                (lambda (l)
                  (if (null? l)
                      '()
                      (cons (f (car l)) ((cmap f) (cdr l))))))])
         ((cmap f) (cons 1 (cons 2 (cons 3 '())))))))
```

```
    (empty-env))
((1 1 2 3) (2 2 3) (3 3))
```