Homework 2b

Jason Hemann

December 6, 2017

1 Preamble

- Do consult class notes, online lecture notes, and test cases when completing this assignment.
- You should not use `letrec` in any question on this part (Part b) of this assignment.
- You should use `pmatch` in each question of this part (Part b) of this assignment
- Load `pmatch.scm` in the top of your file: `(load "pmatch.scm")`
- You may, however, find Scheme's `assv` and `remv` useful.
- You may find some of the functions from Part a of use to you as well.
- You should be writing recursive functions over lambda-calculus expressions.

2 Part 1 : Free, Bound, Lexical Address

1. Define and test a procedure `lambda->lumbda` that takes a lambda-calculus expression and returns the expression unchanged with the exception that you replace each `lambda` keyword with the word `lumbda` (notice you should leave unchanged occurrences of `lambda` as a variable or formal parameter).

```
> (lambda->lumbda 'x)
x
> (lambda->lumbda '(lambda (x) x))
(lumbda (x) x)
> (lambda->lumbda '(lambda (z) ((lambda (y) (a z)) (h (lambda (x) (h a))))))
(lumbda (z) ((lumbda (y) (a z)) (h (lumbda (x) (h a)))))
> (lambda->lumbda '(lambda (lambda) lambda))
(lumbda (lambda) lambda)
> (lambda->lumbda '((lambda (lambda) lambda) (lambda (y) y)))
((lumbda (lambda) lambda) (lumbda (y) y))
> (lambda->lumbda '((lambda (x) x) (lambda (x) x)))
((lumbda (x) x) (lumbda (x) x))
```

2. Define and test a procedure `var-occurs?` that takes a variable name and a lambda-calculus expression and returns a boolean answering whether that variable *occurs* in the expression. Here and

forevermore in this class we use the word "occur" in its technical sense: for us, a formal parameter does not count as a variable occurrence. You should write this as a recursive procedure.

```
> (var-occurs? 'x 'x)
#t
> (var-occurs? 'x '(lambda (x) y))
#f
> (var-occurs? 'x '(lambda (y) x))
#t
> (var-occurs? 'lambda '(lambda (x) y))
#f
> (var-occurs? 'x '((z y) x))
#t
```

3. Define and test a procedure `vars` that takes a lambda-calculus expression and returns a list containing all variables that *occur* in the expression. The order of the variables in your answer does not matter.

```
> (vars 'x)
(x)
> (vars '(lambda (x) x))
(x)
> (vars '((lambda (y) (x x)) (x y)))
(x x x y)
> (vars '(lambda (z) ((lambda (y) (a z))
                      (h (lambda (x) (h a))))))
(a z h h a)
```

4. Define and test a modification of `vars` called `unique-vars` that behaves like `vars` but does not return duplicates. Use `union` in your definition. You should write this as a recursive procedure.

```
> (unique-vars '((lambda (y) (x x)) (x y)))
(x y)
> (unique-vars '((lambda (z) (lambda (y) (z y))) x))
(z y x)
> (unique-vars '((lambda (a) (a b)) ((lambda (c) (a c)) (b a))))
(c b a)
```

5. Define and test a procedure `var-occurs-free?` that takes a symbol and a lambda-calculus expression and returns `#t` if that variable *occurs free* in that expression, and `#f` otherwise. The solution developed in class used a list as an accumulator, your solution should not. You should write this as a recursive procedure.

```
> (var-occurs-free? 'x 'x)
#t
> (var-occurs-free? 'x '(lambda (y) y))
```

```
#f
> (var-occurs-free? 'x '(lambda (x) (x y)))
#f
> (var-occurs-free? 'x '(lambda (x) (lambda (x) x)))
#f
> (var-occurs-free? 'y '(lambda (x) (x y)))
#t
> (var-occurs-free? 'y '((lambda (y) (x y)) (lambda (x) (x y))))
#t
> (var-occurs-free? 'x '((lambda (x) (x x)) (x x)))
#t
```

6. Define and test a procedure `var-occurs-bound?` that takes a symbol and a lambda-calculus expression and returns `#t` if that variable *occurs bound* in the expression, and `#f` otherwise. The solution developed in class used an accumulator, your solution should not. You should write this as a recursive procedure.

```
> (var-occurs-bound? 'x 'x)
#f
> (var-occurs-bound? 'x '(lambda (x) x))
#t
> (var-occurs-bound? 'y '(lambda (x) x))
#f
> (var-occurs-bound? 'x '((lambda (x) (x x)) (x x)))
#t
> (var-occurs-bound? 'z '(lambda (y) (lambda (x) (y z))))
#f
> (var-occurs-bound? 'z '(lambda (y) (lambda (z) (y z))))
#t
> (var-occurs-bound? 'x '(lambda (x) y))
#f
> (var-occurs-bound? 'x '(lambda (x) (lambda (x) x)))
#t
```

7. Define and test a procedure `unique-free-vars` that takes a lambda-calculus expression and returns a list of all the variables that occur free in that expression. Order doesn't matter, but the list must not contain duplicate variables. You may find it helpful to use the definition of `unique-vars` as a starting point. You should write this as a recursive procedure.

```
> (unique-free-vars 'x)
(x)
> (unique-free-vars '(lambda (x) (x y)))
(y)
> (unique-free-vars '((lambda (x) ((x y) e)) (lambda (c) (x (lambda (x) (x (e c)))))))
(y e x)
```

Note that in the third example above,

```
((lambda (x) ((x y) e)) (lambda (c) (x (lambda (x) (x (e c))))))
```

is a single lambda-calculus expression (a procedure application), not a list of lambda-calculus expressions.

8. Define and test a procedure `unique-bound-vars` that takes a lambda-calculus expression and returns a list of all the variables that *occur bound* in the input expression. Order doesn't matter, but the list must not contain duplicate variables.

```
> (unique-bound-vars 'x)
()
> (unique-bound-vars '(lambda (x) y))
()
> (unique-bound-vars '(lambda (x) (x y)))
(x)
> (unique-bound-vars '((lambda (x) ((x y) e)) (lambda (c) (x (lambda (x) (x (e c)))))))
(x c)
> (unique-bound-vars '(lambda (y) y))
(y)
> (unique-bound-vars '(lambda (x) (y z)))
()
> (unique-bound-vars '(lambda (x) (lambda (x) x)))
(x)
```

9. In a subset of Scheme where `lambda` expressions have only one argument, the lexical address of a variable is the number of `lambda`'s between the place where the variable is bound (also known as the formal parameter) and the place where it occurs. For example, in the following expression:

```
(lambda (o)
  (lambda (r)
    (lambda (s)
      (lambda (p)
        (lambda (g)
          o)))))
```

The `o` at the very bottom is a bound occurrence. It has a lexical address of `4`, because there are four `lambda` expressions between the formal parameter `o` at the top and the occurrence of `o` at the bottom.

Define and test a procedure `lex` that takes a lambda-calculus expression and an accumulator (which starts as the empty list), and returns the same expression with all bound variable references replaced by lists of two elements whose `car` is the symbol `var` and whose cadr is the lexical address of the referenced variable. Your solution should not encounter or make reference to unbound variables; we will deem these bad data.

```
> (lex '(lambda (x) x) '())
(lambda (var 0))
> (lex '(lambda (y) (lambda (x) y)) '())
(lambda (lambda (var 1)))
> (lex '(lambda (y) (lambda (x) (x y))) '())
(lambda (lambda ((var 0) (var 1))))
> (lex '(lambda (x) (lambda (x) (x x))) '())
(lambda (lambda ((var 0) (var 0))))
> (lex '(lambda (y) ((lambda (x) (x y)) (lambda (c) (lambda (d) (y c))))) '())
(lambda ((lambda ((var 0) (var 1))) (lambda (lambda ((var 2) (var 1))))))
> (lex '(lambda (a)
          (lambda (b)
            (lambda (c)
              (lambda (a)
                (lambda (b)
                  (lambda (d)
                    (lambda (a)
                      (lambda (e)
                        (((((a b) c) d) e) a)))))))))) '())
(lambda
  (lambda
    (lambda
      (lambda
        (lambda
          (lambda
            (lambda
              (lambda
                (lambda
                  ((((((var 1) (var 3)) (var 5)) (var 2)) (var 0)) (var 1)))))))))))
> (lex '(lambda (a)
          (lambda (b)
            (lambda (c)
              (lambda (w)
                (lambda (x)
                  (lambda (y)
                    ((lambda (a)
                       (lambda (b)
                         (lambda (c)
                           (((((a b) c) w) x) y))))
                     (lambda (w)
                       (lambda (x)
                         (lambda (y)
                           (((((a b) c) w) x) y)))))))))))) '())
(lambda
  (lambda
    (lambda
      (lambda
```

```
(lambda
  (lambda
    ((lambda
       (lambda
         (lambda
           ((((((var 2) (var 1)) (var 0)) (var 5)) (var 4)) (var 3)))))
     (lambda
       (lambda
         (lambda
           ((((((var 8) (var 7)) (var 6)) (var 2)) (var 1)) (var 0))))))))))))
```

You can approach this problem in different ways. My suggestion is to build some `lambda` expressions on pen and paper, and then by hand find the lexical addresses of some variables that occur in them. Try and do it almost mechanically, starting from the top of the expression and working your way down. Then reflect what it is you're doing, and try and figure out how to do it without having to go back up the tree. That is, ensure that when you get to a variable position in the expression where you need to fill in the lexical address, that you already have all the information you need to figure it out.

Although you are not required to use it, you might also find useful the following help function:

```
(define list-index-of-eqv?
  (lambda (x ls)
    (cond
      ((eqv? x (car ls)) 0)
      (else (add1 (list-index-of-eqv? x (cdr ls)))))))
```