**CSSE 304 Assignment 10 Updated for Spring, 2016**

**No input error-checking is required.** You may assume that all arguments have the correct form.
**Abbreviations for the textbook:** EoPL - *Essentials of Programming Languages*, 3rd Edition.

## Mutation is not allowed on this assignment.

**#1** (15 points) `free-vars`, `bound-vars`. LCExp is defined by a grammar on page 9 of EoPL. Given a LcExp *e*, `(free-vars e)` returns the set of all variables that occur free in e. `bound-vars` is similar. Write these procedures directly; do not use `occurs-free` or `occurs-bound` in your definitions. Your code only needs to process the simple lambda-calculus expressions from the grammar from EoPL, not the extended expressions from problem 3 and 4 of this assignment.

```
> (free-vars   '((lambda (x) (x y)) (z (lambda (y) (z y)))))
(y z)
> (bound-vars '((lambda (x) (x y)) (z (lambda (y) (z z)))))
(x)
```

**#2** (40 points) Expand `occurs-free?` and `occurs-bound?` (written in class and in the textbook for basic lambda-calculus expressions) to incorporate the following language features into your code. You can find the original `occurs-free?` and `occurs-bound?` from the textbook at
http://www.rose-hulman.edu/class/csse/csse304/201530/Resources/Code-from-Textbook/1.scm

a) Scheme `lambda` expressions may now have more than one (or zero) parameters, and Scheme procedure calls may have more than one (or zero) arguments. Modify the formal definitions of `occurs-free?` and `occurs-bound?` to allow `lambda` expressions with any number of parameters and procedure calls with any number of arguments. Then modify the procedures `occurs-free?` and `occurs-bound?` to include these new definitions.
b) Extend the formal definitions of `occurs-free?` and `occurs-bound?` to include `if` expressions, and implement these in your code. You are only required to handle `if` expressions that have both a "then" part and an "else" part.
c) Extend the formal definitions of `occurs-free?` and `occurs-bound?` to include Scheme `let` and `let*` expressions, and implement these in your code.
d) Extend the formal definitions of `occurs-free?` and `occurs-bound?` to include Scheme `set!` expressions, and implement these in your code. Note that `set!` does not bind any variables.

```
(occurs-bound? 'x '(lambda (y) (set! x y)))              ➔ #f
(occurs-free?  'y '(lambda (x a b) y))                   ➔ #t
(occurs-free? 'b '(let* ((y a) (x b)) ((x y) z)))        ➔ #t
(occurs-free? 'set! '(lambda (x) (set! x y)))            ➔ #f   ; set! is Scheme syntax, not a variable
(occurs-bound? 'z '(lambda () (let* ((x a) (y x)) (if (y z) (lambda () x) (lambda () y))))) ➔ #f
```

See the test cases for additional examples.

Assignment continues on the next page

**#3** (30 points). `lexical-address`. Write a procedure `lexical-address` that takes an expression like those from the previous problem (except that you are not required to do `let*` expressions for this problem) and returns a copy of the expression with every bound occurrence of a variable `v` replaced by a list `(: d p)`. The two numbers `d` and `p` are the lexical depth and position of that variable occurrence. If the variable occurrence `v` is free, produce the following list instead: `(: free xyz)` To produce the symbols **:** and `free`, use the code **'**`:` and **'**`free`.

**Hint:** It may be easiest to do this with a recursive helper procedure that keeps track of bound variables and their levels as it descends into various levels of the expression.

**Examples:**

```
(lexical-address '(lambda (a b c)
                    (if (eq? b c)
                        ((lambda (c)
                           (cons a c))
                         a)
                        b)))                ➔
(lambda (a b c)
  (if ((: free eq?) (: 0 1) (: 0 2))
      ((lambda (c) ((: free cons) (: 1 0) (: 0 0)))
       (: 0 0))
      (: 0 1)))
```

---

```
(lexical-address
 '((lambda (x y)
     (((lambda (z)
         (lambda (w y)
           (+ x z w y)))
       (list w x y z))
      (+ x y z)))
   (y z)))              ➔
((lambda (x y)
   (((lambda (z)
       (lambda (w y)
         ((: free +) (: 2 0) (: 1 0) (: 0 0) (: 0 1))))
     ((: free list) (: free w) (: 0 0) (: 0 1) (: free z)))
    ((: free +) (: 0 0) (: 0 1) (: free z))))
 ((: free y) (: free z)))
```

---

```
(lexical-address
 '(lambda (a b c)
    (if (eq? b c)
        ((lambda (c) (cons a c))
         a)
        b)))        ➔

(lambda (a b c)
  (if ((: free eq?)(: 0 1) (: 0 2))
      ((lambda (c) ((: free cons) (: 1 0) (: 0 0)))
       (: 0 0))
      (: 0 1)))
```

**#4** (30 points*). `un-lexical-address`. Its input will be in the form of the output from `lexical-address` , as described in the previous problem. When I test it, I will evaluate

`(un-lexical-address (lexical-address <some-expression>))`

and test whether this returns somethi8ng that is `equal?` to the original expression. You cannot get credit for this problem unless you also get a significant number of the points for `lexical-address`. [For example, someone who defined both `lexical-address` and `un-lexical-address` to be the identity procedure would trick the grading program into giving them full credit for `un-lexical-address`, but I would assign their actual grade to be zero points for both problems as after we look at the code by hand.]

Note: `lexical-address` is harder than `un-lexical-address`; if there are errors in your `lexical-address` code, they will most likely be discovered when you test `un-lexical-address`.

**Hint Copied from Piazza (Spring, 2016):** **next page**

# A10b lexical-address hint

I gave this hint verbally in class on both days when we discussed `lexical-address`, but that was a long time ago, so I am reminding you now and giving you a little bit more detail.

`lexical-address` and `un-lexical-address` will each need to have a recursive helper procedure. Each of these procedures will have a parameter that is the current "scope-list". It will be a list of lists of variables, the variables bound by the `lambda`s and `let`s that the current expression is inside of.

For example, consider `(lambda (x y) (lambda (y z) (y (+ x z))))`, When your lexical-address code does the recursive call for the expression `(+ x z)`, the scope-list might be `((y z) (x y))`. A separate "lookup" procedure can use the scope-list to find the lexical depth and position for each local variable. It can also determine that `+` is a free variable, because `+` is not in the scope-list.

When the recursive call is for a non-binding expression (such as `if` or a procedure application), it passes the scope-list unchanged. When it is the body of a `let` or `lambda`, it passes in an expanded scope-list that includes the new bound variables.

`trace` and `trace-lambda` are your friends!

hw