C311/B521/A596 Programming Languages

Home | Policies | Instructors | Resources | Schedule | Assignments | B621

Login

Assignment 2: Free, Bound, and Lexical Address

> There may, indeed, be other applications of the system than its use as a logic.
>
> —*Alonzo Church, writing about the lambda calculus in 1932*

Guidelines for this assignment

- You must test your solutions before submitting your assignment. We have provided some test cases for each exercise to get you started, but the provided test cases are not exhaustive.
- Like last time, we have a test file for you: a2-student-tests.rkt. You may find your order of answers on `vars`, `unique-vars`, `unique-free-vars`, `unique-bound-vars`, or `unify` are different than what we expected. This is totally acceptable.
- You may have the `var-occurs-free?` and `var-occurs-bound?` predicates in your notes from lecture this week. However, *don't* use these versions in this assignment.
- You may, however, find assv and remv useful.
- For the purposes of this assignment, assume that *lambda-calculus expressions* consist of:
  - variables
  - `lambda` expressions that take exactly one argument and have exactly one body
  - applications of two lambda calculus expressions
- Place all of your code in a file named `a2.rkt` and submit it via Oncourse.

Assignment

Part 1: Natural Recursion Refresher

1. Consider the following partial definition of the `list-ref` function. It is intended to operate similarly to Racket's list-ref.

```
(define list-ref
  (lambda (ls n)
    (letrec
      ((nth-cdr
         (lambda (n)
    ;; complete the definition
             )))
      (car (nth-cdr n)))))
```

The body of the function that is the right-hand side of `nth-cdr` is missing. Complete the definition of list-ref with a naturally-recursive implementation of `nth-cdr`, so that the following work correctly.

```
> (list-ref '(a b c) 2)
c
> (list-ref '(a b c) 0)
a
```

Remember, you need not consider bad data in your definition.

2. Define and test a procedure `union` that takes two lists with no duplicates, and returns a list containing the union of the two input lists. You may find it helpful to use Racket's memv for this definition. Again, the order of the elements in your answer does not matter.

```
> (union '() '())
()
> (union '(x) '())
(x)
> (union '(x) '(x))
(x)
> (union '(x y) '(x z))
(x y z)
```

3. Define and test a procedure `extend` that takes two arguments, say `x` and `pred`. The second argument `pred` is a predicate. (Recall what predicates are and how to use them from the previous assignment.) What `extend` returns should be another predicate. The returned predicate should be satisfied exactly by those things that are `eqv?` to `x` or satisfy `pred`.

```
> ((extend 1 even?) 0)
#t
> ((extend 1 even?) 1)
#t
> ((extend 1 even?) 2)
#t
> ((extend 1 even?) 3)
#f
> (filter (extend 1 even?) '(0 1 2 3 4 5))
(0 1 2 4)
> (filter (extend 3 (extend 1 even?)) '(0 1 2 3 4 5))
(0 1 2 3 4)
> (filter (extend 7 (extend 3 (extend 1 even?))) '(0 1 2 3 4 5))
(0 1 2 3 4)
```

4. Define and test a procedure `walk-symbol` that takes a symbol `x` and an association list `s`. An *association list* is a list of pairs of associated values. For example, the following is an association list:

```
((a . 5) (b . (1 2)) (c . a))
```

Your procedure should search through `s` for the value associated with `x`. If the associated value is a symbol, it too must be walked in `s`. If `x` has no association, then `walk-symbol` should return `x`.

```
> (walk-symbol 'a '((a . 5)))
5
> (walk-symbol 'a '((b . c) (a . b)))
c
> (walk-symbol 'a '((a . 5) (b . 6) (c . a)))
5
> (walk-symbol 'c '((a . 5) (b . (a . c)) (c . a)))
5
> (walk-symbol 'b '((a . 5) (b . ((c . a))) (c . a)))
((c . a))
> (walk-symbol 'd '((a . 5) (b . (1 2)) (c . a) (e . c) (d . e)))
5
> (walk-symbol 'd '((a . 5) (b . 6) (c . f) (e . c) (d . e)))
f
```

Part 2: Free, Bound, Lexical Address

You must **use** pmatch **in** each **of the remaining problems. The brainteasers might be easier with it as well.** You may find some of the functions from Part 1 of use to you as well. For the most part, you should expect to be performing recursion on lambda-calculus expressions. If you have correctly installed our libraries, you should need only `(require C311/pmatch)` immediately below `#lang racket` in your file.

5. Define and test a procedure `lambda->lumbda` that takes a lambda-calculus expression and returns the expression unchanged with the exception that each `lambda` as a keyword has been replaced with the word `lumbda` (notice occurrences of `lambda` as a variable should be left alone).

```
> (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))
```

6. Define and test a procedure `vars` that takes a lambda-calculus expression and returns a list containing all variables that occur in the expression. This should be a straightforward modification of `lambda->lumbda`, and 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)
```

7. Define and test a modification of `vars` called `unique-vars` that behaves like `vars` but does not return duplicates. Use `union` in your definition.

```
> (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)
```

8. 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.

```
> (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? '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
```

9. 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.

```
> (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
```

10. 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 not use `var-occurs-free?` or `var-occurs-bound?` as helpers.

```
> (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.

11. 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) (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 (x) y))
()
> (unique-bound-vars '(lambda (x) (y z)))
()
```

12. In a subset of Racket where `lambda`s 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 1) 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, 2) free variables similarly wrapped within a list whose `car` is the symbol `free-var` and whose `cadr` is the free variable, and 3) the (now superfluous) formal parameters of the `lambda` expressions are dropped.

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

Brainteasers

13. Here is the `walk*` procedure, which takes a term `v` and an association list `s`:

```
(define walk*
  (lambda (v s)
    (let ((v (cond ((symbol? v) (walk-symbol v s))
                   (else v))))
      (cond ((pair? v)
             (cons (walk* (car v) s)
                   (walk* (cdr v) s)))
            (else v)))))
```

We say that two terms `u` and `v` are *equivalent* relative to `s` if any of the following properties hold:

- `u` and `v` are `equal?`, or
- if we were to replace all the symbols in `u` and `v` with the result of calling `walk*` on them, the resulting values would be `equal?`, or
- we can add associations to `s` to make `u` and `v` equivalent with respect to the extended association list.

Define and test a procedure `unify` that takes two terms `u` and `v` and an association list `s`. `unify` behaves as follows: it returns `#f` if `u` and `v` are not equivalent; otherwise, it returns `s` or a possibly extended association list in which `u` and `v` are equivalent. When you are finished, you will have a procedure that performs unification.

One thing to keep in mind is that you cannot have a constant (such as 5) on the left-hand side of a pair in `s`. You can only have variables (such as `x`) on the left-hand side of a pair. Two different constants such as 5 and 6 can never be equivalent, no matter what `s` is. A logic programmer would say, “5 can never unify with 6.”

```
> (unify 'x 'y '())
((x . y)) ;; Another correct answer: ((y . x))
> (unify '(x) '(y) '())
((x . y)) ;; Another correct answer: ((y . x))
> (unify 5 5 '())
()
> (unify 5 6 '())
#f
> (unify '(5 6) '(x y) '()) ;; Another correct answer: ((y . 6) (x . 5))
((x . 5) (y . 6))
> (unify '(z 5) '(5 x) '((z . 3) (x . z)))
#f
> (unify '((x . 5) (y . z)) '((y . 5) (x . 5)) '((z . 5) (x . y))) ;; Another correct answer: (
((z . 5) (x . y))
```

14. Consider again the scenario of the `walk-symbol` problem. Imagine that we frequently look up values in that association list. Walking the full chain every time may become prohibitively expensive, as certain perverse chains may be arbitrarily long. Consider the work you would have to do to walk `a` twice in the following association list.

```
'((z . 26) (y . z) (x . y) ... (b . c) (a . b))
```

To partially alleviate this burden, we will implement `walk-symbol-update` with *path-compression*, in the following manner. We will write our association list such that the right-hand side of each association is always a box that contains a value. Boxes are mutable memory references, meaning we can change the value the box contains. Then, when we walk the association list to find the final value for the symbol we started with, we can also change the values in boxes we had to walk through along the way, so that the right-hand side of each of those also contains the final value. Thus, if we have to walk that same symbol again, the lookup will be faster. See the following example.

```
> (define a-list `((c . ,(box 15)) (e . ,(box 'f)) (b . ,(box 'c)) (a . ,(box 'b))))
> a-list
((c . #&15) (e . #&f) (b . #&c) (a . #&b))
> (walk-symbol-update 'a a-list)
15
> a-list
((c . #&15) (e . #&f) (b . #&15) (a . #&15))
> (walk-symbol-update 'a a-list)
15
> a-list
((c . #&15) (e . #&f) (b . #&15) (a . #&15))
```

Without boxes (or some side-effect) we would have been required to re-copy the entire data structure each time we wanted to change a portion of it. You will find it useful to consult the Racket Documentation about boxes for information about the `box`, `unbox`, and `set-box!` functions for this problem.

Just Dessert

15. A variable can both occur free and occur bound in the same expression. Define a predicate `var-occurs-both?` that takes a variable `x` and a lambda-calculus expression, and returns `#t` if `x` occurs both free and bound within that expression. Otherwise, `var-occurs-both?` returns `#f`. Your solution should be a **one-pass solution**, meaning you should not recur over the same data twice, and you should not use an accumulator.

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