Login

Assignment 4: Dynamic Scope

> On two occasions I have been asked [by members of Parliament],–"Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" [...]

Assignment

This assignment has three parts. In addition to an improved `lex`, we expect you to turn in **three interpreters**: `value-of-fn`, `value-of-ds`, and a third, `value-of-dynamic` with your implementation of dynamic scope. H311/B521 students will have a fourth interpreter.

You should be able to use the a4-student-tests.rkt file to test your solutions.

```
> (require "a4-student-tests.rkt")
> (test-file #:file-name "a4.rkt")
...
```

and that should get you going. Of course, **these tests are not exhaustive; you should add your own tests as well**.

Part I

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 fairly straightforward extension, but it also serves as a chance to improve a misbehaving `lex` from Assignment 2. In order to better 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)))
```

Part II

For this part of the assignment, use **one** of your interpreters from last week's assignment as a starting point. You may pick either `value-of`, `value-of-fn` or `value-of-ds`. But for Part II you should pick, and stick with, a *single* representation of environments. If you choose to start with `value-of`, leave your environments representation dependent using higher-order functions. If you chose to start with `value-of-fn` change either all three of `empty-env-fn`, `apply-env-fn`, and `extend-env-fn` to `empty-env`, `apply-env`, and `extend-env`. If you chose to start with `value-of-ds` change all three of `empty-env-ds`, `apply-env-ds`, and `extend-env-ds`, to `empty-env`, `apply-env`, and `extend-env`. In either of the latter two cases, use these new names for the environment helpers in your interpreter. Having done so, create two *new* interpreters that are representation independent with respect to closures: `value-of-fn` and `value-of-ds`, respectively.

```
1. ''value-of-fn'' should use a functional representation of closures.
2. ''value-of-ds'' should use a data-structural representation of closures.
```

You should write two new closure helper functions for each of your interpreters. Write `apply-closure-fn` and `closure-fn` for `value-of-fn`, and write `apply-closure-ds`, and `closure-ds` for `value-of-ds`.

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**.

```
> (value-of-fn
    '((lambda (x) (if (zero? x)
                      12
                      47))
       0)
    (empty-env))
12
> (value-of-fn
   '(let ([y (* 3 4)])
      ((lambda (x) (* x y)) (sub1 6)))
   (empty-env))
60
> (value-of-fn
   '(let ([x (* 2 3)])
```

```
      (let ([y (sub1 x)])
        (* x y)))
   (empty-env))
30
> (value-of-fn
   '(let ([x (* 2 3)])
      (let ([x (sub1 x)])
        (* x x)))
   (empty-env))
25
> (value-of-ds
    '((lambda (x) (if (zero? x)
                      12
                      47))
       0)
    (empty-env))
12
> (value-of-ds
   '(let ([y (* 3 4)])
      ((lambda (x) (* x y)) (sub1 6)))
   (empty-env))
60
> (value-of-ds
   '(let ([x (* 2 3)])
      (let ([y (sub1 x)])
        (* x y)))
   (empty-env))
30
> (value-of-ds
   '(let ([x (* 2 3)])
      (let ([x (sub1 x)])
        (* x x)))
   (empty-env))
25
```

Part III

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

Explanation of dynamic scope

The interpreters we have been writing so far have been implemented in such a way 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. This technique is called *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`.

As you can see, dynamic scope is a little strange, but it does have its uses.

Define `value-of-dynamic`, an interpreter that implements dynamic scope. You can start with the dynamically-scoped interpreter we wrote in class that used `match-let`. You should be able to share your environment helpers from Parts I and II above, 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))
```

Brainteasers

4. We've been talking a whole lot about representation independence. It would sure be nice to have a single interpreter where we could just pass in the various helper functions. That way we could write the interpreter once, pass in implementations of our environment and closure helpers, and then get an interpreter that will just take the expression we want to evaluate, that uses those helpers. From such a definition, it is obvious on first inspection of this single interpreter is indeed representation independent with respect to both environments and closures.

So let's do it.

Write a single function named `value-of-ri` that'll take `empty-env`, `extend-env`, `apply-env`, `closure`, and `apply-closure`, and return an interpreter expecting a single expression. You'll need to pass an additional parameter to your closure helper functions, so define them as `closure-fn-ri` and `apply-closure-fn-ri closure-ds-ri` and `apply-closure-ds-ri`, and make sure they take this additional parameter. Go ahead and include your regular `if`, `*`, `sub1`, `zero?`, `let`, forms, along with numbers and booleans, and `lambda`-calculus expressions. **You should not pass your helper functions to recursive calls.** Here's what the calls to initially kick off the interpreter should look like.

```
>((value-of-ri empty-env-fn extend-env-fn apply-env-fn closure-fn-ri apply-closure-fn-ri) '((lambda (x) x) 5))
5
>((value-of-ri empty-env-ds extend-env-ds apply-env-ds closure-ds-ri apply-closure-ds-ri) '((lambda (x) x) 5))
5
>((value-of-ri empty-env-fn extend-env-fn apply-env-fn closure-ds-ri apply-closure-ds-ri) '((lambda (x) x) 5))
5
```

```
>((value-of-ri empty-env-ds extend-env-ds apply-env-ds closure-fn-ri apply-closure-fn-ri) '((lambda (x) x) 5))
5
```

Your solution should involve a letrec.

Just Dessert

5. Our `value-of` uses an environment as a way to accumulate the associations of free variables. As a result of this decision to accumulate the associations, if we wanted to implement lambda abstractions correctly, we had to make the value of a lambda abstraction not just code, but code together with an environment (which we term a closure). This means that when we evaluate a lambda abstraction, the value **definitely** isn't just that lambda abstraction. For our a3 interpreter, we'd get something like the following:

```
> (value-of '(lambda (x) x) (lambda (y) (error "unbound identifier ~s~n" y)))
<#procedure>
```

That's sorta annoying. If we implement a function that evaluates expressions in a naturally-recursive style, then we won't need an environment, and so we can return plain code instead of building closures. We may have mentioned that implementing `value-of` the natural-recursive style is fraught with peril, because implementing substitution is tricky. Instead, we think of the language interpreted by `value-of` as a set of symbolic terms, though, we don't concern ourselves representations of procedures or environments. Instead, we just rewrite one term to another term according to some rules. For example:

```
((lambda (x) x) (lambda (y) y))
-> (lambda (y) y)
```

Here's how it is usually done. We would first define our main rewriting rule, `beta-n`:

```
((lambda (x) e1) e2) beta-n e1[e2/x]
```

This relation tells us that when we have an expression of the form:

```
((lambda (x) e1) e2)
```

We can rewrite it to the expression `e1` where `e2` has been substituted for `x`. This substitution is not trivial, however, since it must not change the scope of variables involved in the substitution. So, we define e1[e2/x] as:

```
               x1[e/x1] = e
               x2[e/x1] = x2 if x1 is not x2
(lambda (x1) e1)[e2/x1] = (lambda (x1) e1)
(lambda (x1) e1)[e2/x2] = (lambda (x3) e1[x3/x1][e2/x2])
                          if x1 is not x2,
                             x2 is not x3,
                             x3 is not a free variable in (lambda (x1) e1)
```

```
                              and x3 is not a free variable in e2
         (e1 e2)[e3/x] = (e1[e3/x] e2[e3/x])
```

Perilous indeed.

There are other rewriting rules for constants as well as for deciding where in a nested expression to apply the next rewriting rule. We aren't providing those here, but we are including them in the provided starter code. For another description of substitution, see *(Formal Syntax and Semantics of Programming Languages [http://www.divms.uiowa.edu/~slonnegr/plf/Book/], Ken Slonneger and Barry L. Kurt, Addison-Wesley Publishing Company, 1995. 146 - 149.)*: sub.pdf

Let's try it another way. Instead of going through a careful substitution process precisely where we need to do it, we're going to go hog wild. `alpha-all` takes an expression and changes all bound variables in the expression, and the formal parameters that bind them, to brand new variables. This way we do substitution, we won't accidentally induce any capture.

```
(define alpha-all
  (lambda (exp)
    (match exp
      [`,x #:when (symbol? x) x]
      [`(lambda (,x) ,body)
       (let ((g (gensym (symbol->string x))))
         `(lambda (,g) ,(subst g x (alpha-all body))))]
      [`(,rator ,rand)
       `(,(alpha-all rator) ,(alpha-all rand))])))
```

The definition of `alpha-all` relies on `subst`. Your mission is simply to implement `subst`. The `subst` function performs naive substitution.

With a suitably-implemented `subst`, we can implement a reducer that evaluates lambda-calulus expressions to values equivalent to those computed by `value-of`.

```
(define reducer
  (lambda (under before)
    (lambda (str)
      (letrec
        ((reducer (lambda (exp)
           (match exp
             [`,x #:when (symbol? x) x]
             [`(lambda (,x) ,body)
              `(lambda (,x) ,((under reducer) body))]
             [`(,rator ,rand)
              (match (reducer rator)
                [`(lambda (,x) ,body)
                 (str (subst ((before reducer) rand) x (alpha-all body)))]
                [`,else `(,else ,((before reducer) rand))])])))
         reducer))))
```

```
(define yes (lambda (f) (lambda (x) (f x))))
(define no (lambda (f) (lambda (x) x)))

(define by-value (reducer no yes))
```

But equally as interesting, by making different choices as to when we perform reductions, we can generate a whole range of reduction strategies.

```
(define applicative (reducer yes yes))
(define head-spine (reducer yes no))
(define by-name (reducer no no))

(define ao-nf
  (letrec ((str (lambda (exp) ((applicative str) exp))))
    str))

(define bv-wnf
  (letrec ((str (lambda (exp) ((by-value str) exp))))
    str))

(define he-hnf
  (letrec ((str (lambda (exp) ((head-spine str) exp))))
    str))

(define bn-whnf
  (letrec ((str (lambda (exp) ((by-name str) exp))))
    str))

(define no-nf (applicative bn-whnf))
(define ha-nf (applicative bv-wnf))
(define hn-nf (applicative he-hnf))
```

For more information on the foregoing, you should consult Sestoft's Demonstrating Lambda Calculus Reduction [http://www.itu.dk/~sestoft/papers/mfps2001-sestoft.pdf].