Login

Assignment 3: Environments and Interpreters

> Inverting the adage that a data type is just a simple programming language, we take the position that a programming language is, semantically, just a complex data type; evaluation of a program is just another operation in the data type.

Assignment

Testing your homeworks

We have provided a suite of test cases your interpreter must pass. To run these tests, you must download a3-student-tests.rkt test file, and do the following.

```
> (require "a3-student-tests.rkt")
> (test-file #:file-name "a3.rkt")
...
```

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

Notes

As you proceed on this assignment, you may find the following Notes on representation independence [https://github.com/mvcccccc/C311Sp19/blob/master/notes/ri_interpreter.pdf] to be of use.

Part 1: Interpreters and Environments

In recent lectures, we've learned how to write an interpreter that takes a Racket expression and returns the result of evaluating that expression. We have also learned to make it representation independent with respect to environments. In fact, we have written two different representations for the helpers `extend-env`, `apply-env`, and `empty-env`. In this assignment, we will implement **four** interpreters. During Part 1, we will implement 3 interpreters: the three interpreters presented in lecture, along with the two different sets of environments helpers for the 2nd and 3rd interpreter. Part 2 implements a new interpreter for a language that you haven't seen before. Place your code for **all four interpreters** in a file named `a3.rkt` and submit it to Oncourse.

- **You must define *two* sets of environment helpers: one that uses functional (higher-order) representation of environments, and one that uses data-structural representation of environments.** Call the representation-dependent version `value-of`, the version with functional helpers `value-of-fn`, and the version with data-structural helpers `value-of-ds`. Your data structures should be the **tagged list** representation demonstrated in class. Notice these names may be different from those presented in lecture. This is a framework for how you should name your procedures and helpers:

```
(define value-of ...)

(define value-of-fn ...)
(define empty-env-fn ...)
(define extend-env-fn ...)
(define apply-env-fn ...)

(define value-of-ds ...)
(define empty-env-ds ...)
(define extend-env-ds ...)
(define apply-env-ds ...)
```

- Your interpreter must handle the following forms: numbers, booleans, variables, lambda-abstraction, application, `zero?`, `sub1`, `*`, `if`, and `let`.
- Remember, your solutions should be compositional.
- You may have seen the expansion of `(let ([x e]) body)` as `((lambda (x) body) e)`. However, when you have a handle on the environment, you can implement `let` in its own right. Therefore, you must not use `lambda` in this way for your interpreter's line for `let` expressions.
- As usual, brainteasers are required for 521 and H311 students.
- These are *not* at all required reading, but for a wider understanding of what we're doing, see Reynolds [http://surface.syr.edu/cgi/viewcontent.cgi?article=1012&context=lcsmith_other] and Danvy [http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.164.8417&rep=rep1&type=pdf].

```
> (value-of
   '((lambda (x) (if (zero? x)
                     #t
                     #f))
     0)
   (lambda (y) (error 'value-of "unbound variable ~s" y)))
#t
> (value-of
   '((lambda (x) (if (zero? x)
                     12
                     47))
     0)
   (lambda (y) (error 'value-of "unbound variable ~s" y)))
12
> (value-of
   '(let ([y (* 3 4)])
      ((lambda (x) (* x y)) (sub1 6)))
   (lambda (y) (error 'value-of "unbound variable ~s" y)))
60
> (value-of
   '(let ([x (* 2 3)])
      (let ([y (sub1 x)])
        (* x y)))
   (lambda (y) (error 'value-of "unbound variable ~s" y)))
```

```
30
> (value-of
   '(let ([x (* 2 3)])
      (let ([x (sub1 x)])
        (* x x)))
   (lambda (y) (error 'value-of "unbound variable ~s" y)))
25
> (value-of
   '(let ((! (lambda (x) (* x x))))
      (let ((! (lambda (n)
                 (if (zero? n) 1 (* n (! (sub1 n)))))))
        (! 5)))
   (lambda (y) (error 'value-of "unbound variable ~s" y)))
80
> (value-of
   '(((lambda (f)
        (lambda (n) (if (zero? n) 1 (* n ((f f) (sub1 n))))))
      (lambda (f)
        (lambda (n) (if (zero? n) 1 (* n ((f f) (sub1 n)))))))
     5)
   (lambda (y) (error 'value-of "unbound variable ~s" y)))
120
> (value-of-fn
   '((lambda (x) (if (zero? x)
                     #t
                     #f))
     0)
   (empty-env-fn))
#t
> (value-of-fn
   '((lambda (x) (if (zero? x)
                     12
                     47))
     0)
   (empty-env-fn))
12
> (value-of-fn
   '(let ([y (* 3 4)])
      ((lambda (x) (* x y)) (sub1 6)))
   (empty-env-fn))
60
> (value-of-fn
   '(let ([x (* 2 3)])
      (let ([y (sub1 x)])
        (* x y)))
   (empty-env-fn))
30
> (value-of-fn
```

```
   '(let ([x (* 2 3)])
      (let ([x (sub1 x)])
        (* x x)))
   (empty-env-fn))
25
> (value-of-fn
   '(((lambda (f)
        (lambda (n) (if (zero? n) 1 (* n ((f f) (sub1 n))))))
      (lambda (f)
        (lambda (n) (if (zero? n) 1 (* n ((f f) (sub1 n)))))))
     5)
   (empty-env-fn))
120
> (value-of-ds
   '((lambda (x) (if (zero? x)
                     #t
                     #f))
     0)
   (empty-env-ds))
#t
> (value-of-ds
   '((lambda (x) (if (zero? x)
                     12
                     47))
     0)
   (empty-env-ds))
12
> (value-of-ds
   '(let ([y (* 3 4)])
      ((lambda (x) (* x y)) (sub1 6)))
   (empty-env-ds))
60
> (value-of-ds
   '(let ([x (* 2 3)])
      (let ([y (sub1 x)])
        (* x y)))
   (empty-env-ds))
30
> (value-of-ds
   '(let ([x (* 2 3)])
      (let ([x (sub1 x)])
        (* x x)))
   (empty-env-ds))
25
> (value-of-ds
   '(((lambda (f)
        (lambda (n) (if (zero? n) 1 (* n ((f f) (sub1 n))))))
      (lambda (f)
```

```
        (lambda (n) (if (zero? n) 1 (* n ((f f) (sub1 n)))))))
     5)
   (empty-env-ds))
120
```

Part 2 "fo-eulav"

4. Having successfully completed the first part of the assignment, you are now well prepared to implement `fo-eulav`. Sample runs are provided below. Only forms present in those sample runs are required.

```
> ;; Ppa
(fo-eulav '(5 (x (x) adbmal)) (lambda (y) (error 'fo-eulav "unbound variable ~s" y)))
5
> ;; Stnemugra sa Snoitcnuf
(fo-eulav '(((x 1bus) (x) adbmal) ((5 f) (f) adbmal)) (lambda (y) (error 'fo-eulav "unbound variable ~s" y)))
4
> ;; Tcaf
(fo-eulav   '(5
 (((((((n 1bus) (f f)) n *) 1 (n ?orez) fi)
                                    (n) adbmal)
                                      (f) adbmal)
  ((((((n 1bus) (f f)) n *) 1 (n ?orez) fi)
                                   (n) adbmal)
                                     (f) adbmal)))
 (lambda (y) (error 'fo-eulav "unbound variable ~s" y)))
120
```

Brainteasers

5. Extend your interpreter `value-of` to support `set!` and `begin2`, where `begin2` is a variant of Racket's `begin` that takes exactly two arguments, and `set!` mutates variables.

```
> (value-of
    '(* (begin2 1 1) 3)
    (lambda (y) (error 'value-of "unbound variable ~s" y)))
3
> (value-of
    '((lambda (a)
        ((lambda (p)
           (begin2
             (p a)
             a))
```

```
         (lambda (x) (set! x 4))))
      3)
     (lambda (y) (error 'value-of "unbound variable ~s" y)))
3
> (value-of
   '((lambda (f)
       ((lambda (g)
          ((lambda (z) (begin2
                        (g z)
                        z))
           55))
        (lambda (y) (f y)))) (lambda (x) (set! x 44)))
   (lambda (y) (error 'value-of "unbound variable ~s" y)))
55
> (value-of
    '((lambda (x)
        (begin2 (set! x 5) x))
      6)
    (lambda (y) (error 'value-of "unbound variable ~s" y)))
5
> (value-of
    '(let ((a 3))
        (begin2 (begin2 a (set! a 4)) a))
    (lambda (y) (error 'value-of "unbound variable ~s" y)))
4
> (value-of
    '((lambda (x)
        (begin2
          ((lambda (y)
             (begin2
               (set! x 0)
               98))
           99)
          x))
      97)
    (lambda (y) (error 'value-of "unbound variable ~s" y)))
0
> (value-of
    '((lambda (y)
        (let ((x (begin2
                   (set! y 7)
                   8)))
          (begin2
            (set! y 3)
              ((lambda (z) y)
               x))))
      4)
    (lambda (y) (error 'value-of "unbound variable ~s" y)))
```

```
3
> (value-of
    '(let ((a 5))
       (let ((y (begin2 (set! a (sub1 a)) 6)))
         (begin2
           (* y y)
           a)))
    (lambda (y) (error 'value-of "unbound variable ~s" y)))
4
```

6. Consider the following interpreter for a deBruijnized version of the lambda-calculus (i.e. lambda-calculus expressions using lexical addresses instead of variables). Notice this interpreter is representation-independent with respect to environments. There are a few other slight variations in the syntax of the language. These are of no particular consequence.

```
(define value-of-lex
  (lambda (exp env)
    (match exp
      [`(const ,expr) expr]
      [`(mult ,x1 ,x2) (* (value-of-lex x1 env) (value-of-lex x2 env))]
      [`(zero ,x) (zero? (value-of-lex x env))]
      (`(sub1 ,body) (sub1 (value-of-lex body env)))
      (`(if ,t ,c ,a) (if (value-of-lex t env) (value-of-lex c env) (value-of-lex a env)))
      (`(var ,num) (apply-env-lex env num))
      (`(lambda ,body) (lambda (a) (value-of-lex body (extend-env-lex a env))))
      (`(,rator ,rand) ((value-of-lex rator env) (value-of-lex rand env))))))

(define empty-env-lex
  (lambda () '()))
```

From the following call one can see we're using a data-structure representation of environments.

```
> (value-of-lex '((lambda (var 0)) (const 5)) (empty-env-lex))
5
```

Define `apply-env-lex` and `extend-env-lex`. Your definitions should be short; in fact, they **should not use `lambda`**. As ever, you are not required to handle bad data. Make sure to add `value-of-lex` and `empty-env-lex` to your file, so that we can test your helpers.

Just Dessert

7. The lambda calculus can be used to define a representation of natural numbers, called Church numerals, and arithmetic over them. For instance, `c5` is the definition of the Church numeral for 5.

```
> (define c0 (lambda (f) (lambda (x) x)))
> (define c5 (lambda (f) (lambda (x) (f (f (f (f (f x))))))))
> ((c5 add1) 0)
5
> ((c0 add1) 0)
0
```

The following is a definition for Church plus, which performs addition over Church numerals.

```
> (define c+ (lambda (m)
               (lambda (n)
                 (lambda (a) (lambda (b) ((m a) ((n a) b)))))))
> (let ((c10 ((c+ c5) c5)))
    ((c10 add1) 0))
10
```

One way to understand the definition of `c+` is that it, when provided two Church numerals, returns a function that, when provided a meaning for `add1` and a meaning for zero, uses provides to `m` the meaning for `add1` and, instead of the meaning for zero, provides it the meaning for its second argument. `m` is the sort of thing that will count up m times, so the result is the meaning of m + n.

Your task, however, is to implement `csub1`, Church predecessor. The following tests should pass.

```
> (((csub1 c5) add1) 0)
4
> (((csub1 c0) add1) 0)
0
```

In the second case, the Church predecessor of Church zero is zero, as we haven't a notion of negative numbers.

This was a difficult problem, but it's fun, so don't Google it. If you think it might help though, consider taking a trip to the dentist [http://link.springer.com/chapter/10.1007%2FBFb0062850].