

C311/B521/A596 Programming Languages

[Home](#)
[Policies](#)
[Instructors](#)
[Resources](#)
[Schedule](#)
[Assignments](#)
[B621](#)
[Login](#)



Assignment 3: Environments and Interpreters

It (the computer) is a medium that can dynamically simulate the details of any other medium, including media that cannot exist physically. It is not a tool, although it can act like many tools.

—Alan Kay

Assignment

Testing your homeworks

We have provided a suite of test cases your interpreter must pass. To run these tests, you must download the  C311.zip archive and  a3-student-tests.rkt test file, do the following.

1. Download and unzip the C311 course archive to a permanent location.
2. In that location run `'raco link </path/to/C311/>'`.
3. Upon doing so, you should be able to, within Racket:

```
> (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.**

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, and two different representations for the helpers. Your task for this assignment is to implement these interpreters presented in lecture, along with the helpers for environments, in addition to a fourth interpreter for a language you have not seen before. Place your code for **all four interpreters** in a file named `a3.rkt` and submit it to Ouncourse.

- 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`. 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.

```
> (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)
    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)
    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))

```

```

    (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 Binary Functions

4. You should add to your representation **dependent** interpreter, `value-of`, forms which handle functions of two arguments and applications of two arguments. As this too must be implemented compositionally, you cannot simply implement it as nested calls to the lambda-abstraction and application forms you've defined previously.

```

> (value-of
  '(((lambda (x y) (* x y)) (* 2 3) (* 4 1))
  (lambda (y) (error 'value-of "unbound variable ~s" y)))
24

```

Part 3 "fo-eulav"

5. 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. You can use the same `empty-env` as above.

```

> ;; Ppa
(fo-eulav '(5 (x (x) adbm1)) (empty-env))
5
> ;; Stnemugra sa Snoitcnuf
(fo-eulav '(((x lbus) (x) adbm1) ((5 f) (f) adbm1)) (empty-env))
4
> ;; Tcaf
(fo-eulav '(5
  ((((((n lbus) (f f)) n *) 1 (n ?orez) fi)
      (n) adbm1)
    (f) adbm1)
  ((((((n lbus) (f f)) n *) 1 (n ?orez) fi)
      (n) adbm1)
    (f) adbm1)))) (empty-env))
120

```

Brainteasers

6. 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!` implements performing side-effects.

```

(test "set!-1"
  (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)
  (test "set!-2"
    (value-of
      '(((lambda (f)
          ((lambda (g)
              ((lambda (z) (begin2
                  (g z)
                  z))
                55))
            (lambda (y) (f y)))) (lambda (x) (set! x 4))))
        (lambda (y) (error 'value-of "unbound variable ~s" y)))
        55)
    (test "set!-3"
      (value-of
        '(((lambda (x)
            (begin2 (set! x 5) x))
          (set! x 5) x))
          (set! x 5) x))
        (set! x 5) x))
        55)
    (test "set!-3"
      (value-of
        '(((lambda (x)
            (begin2 (set! x 5) x))
          (set! x 5) x))
          (set! x 5) x))
        (set! x 5) x))
        55)

```

```

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

```

7. 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.

```

(define value-of-lex
  (lambda (exp env)
    (pmatch exp
      (`,c (guard (or (boolean? c) (number? c))) c)
      (`,(sub1 ,body) (sub1 (value-of-lex body env)))
      (`,(zero? ,body) (zero? (value-of-lex body env)))
      (`,`(* ,n1 ,n2) (* (value-of-lex n1 env) (value-of-lex n2 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)) 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.

Just Dessert

8. As mentioned in class, 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.

We've heard 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](#).

assignment-3.txt · Last modified: 2014/01/30 20:18 by ihmenn

