UMass Boston Computer Science

CS450 High Level Languages

Implementing Recursion, Mutation

Thursday, May 1, 2025

Logistics

- HW 12 out
  - due: Tues 5/6 11am EST

(need “lambda” for hw12)

(don’t need “recursive bind” for hw12)

Last Time

bind, lm, and their environments

```
(bind [x 10]
 (bind [y 20]
  (+ x y)))
```

run

```
(bind [x 10]
 (bind [y 20]
  (+ x y)))
```

Env:
x -> 10
y -> 20

```
(bind [x 10]
 (lm (y)
  (+ x y)))
```

run

```
(lm-result
  '(y)
  (parse '(+ x y))
  )
```

Env: x -> 10

```
( (bind [x 10]
    (lm (y)
     (+ x y)) 20 )
```

run

```
( (bind [x 10]
    (lm (y)
     (+ x y)) 20 )
```

Env:
x -> 10
y -> 20

Last Time

bind, lm, and their environments

```
(bind [x 10]
 (bind [y 20]
  (+ x y)))
```

run

```
(bind [x 10]
 (bind [y 20]
  (+ x y)))
```

Env:
x -> 10
y -> 20

```
(bind [x 10]
 (lm (y)
  (+ x y)))
```

run

```
(lm-result
  '(y)
  (parse '(+ x y))
  )
```

Env: x -> 10

```
( (bind [x 10]
   (lm (y)
    (+ x y)) 20 )
```

run

```
( (bind [x 10]
   (lm (y)
    (+ x y)) 20 )
```

Env:
x -> 10
y -> 20

Last Time

bind = lm + fn call!

```
(bind [x 10]
 (bind [y 20]
  (+ x y)))
```

run

```
(bind [x 10]
 (bind [y 20]
  (+ x y)))
```

Env:
x -> 10
y -> 20

```
(bind [x 10]
 (lm (y)
  (+ x y)))
```

run

```
(lm-result
  '(y)
  (parse '(+ x y))
  )
```

Env: x -> 10

```
( (bind [x 10]
   (lm (y)
    (+ x y)) 20 )
```

run

```
( (bind [x 10]
   (lm (y)
    (+ x y)) 20 )
```

Env:
x -> 10
y -> 20

```
(bind [x e] body)
```

Equivalent!

```
((lm (x) body) e)
```

run

Env:
x -> (run e)

```
(bind [x e] body)
```

run

```
((lm (x) body) e)
```

Env:
x -> (run e)

Running **bind**

```
(define (run p)

  (define (run/env p old-env)
    (match p
      …

     [(bind x e body)
      (define env/x
        (env-add old-env x (run/env e old-env)))
      (run/env body env/x)]

      … ))

 (run/env p INIT-ENV))
```

Read: “environment with x”

Running **lm** + fn call

```
(map (curryr run/env old-env) args)
```

```
(define (450apply fn arg-results)
 (match fn
    …

  [(lm-result params body saved-env)
   (define env/args
     (foldl env-add saved-env params arg-results))]
   (run/env body env/args)]

    … ))
```

Previously

"bind" in "CS450" Lang

```
;; A Variable (Var) is a Symbol
```

```
;; A Prog is one of:
;; …
;; - Var
;; - `(bind [,Var ,Prog] ,Prog)
;; …
```

Reference a variable binding

new binding is in-scope (can be referenced) **here**

Create new variable binding

new binding is **not** in-scope here

Previously

bind examples

```
;; A Prog is one of:
;; …
;; - Var
;; - `(bind [,Var ,Prog] ,Prog)
;; …
```

new binding is **not** in-scope here

```
(check-equal?
  (eval450
   '(bind [x (+ x 20)]
      x))
   UNDEFINED-ERROR )
```

new binding is **not** in-scope here

bind examples, with functions

```
;; A Prog is one of:
;; …
;; - Var
;; - `(bind [,Var ,Prog] ,Prog)
;; - `(lm ,List<Var> ,Prog)
;; - (cons Prog List<Prog>)
;; …
```

```
(check-equal?
  (eval450
   '(bind [f (lm (x) (+ x 4))]
     (f 6)))
  10 )
```

f not in-scope here
(so function can't be recursive!)

```
(check-equal?
  (eval450
   '(bind [f (lm (x) (f x))]
     (f 6)))
  UNDEF-ERR)
```

f not in-scope here
(so function can't be recursive!)

“bind/rec” in “CS450” Lang

```
;; A Prog is one of:
;; …
;; - `(bind/rec [,Var ,Prog] ,Prog)
;; …
```

new binding is in-scope (can be referenced) here

Create new variable binding

new binding is **also** in-scope here!

Racket recursive function examples

Recursive call

RACKET

```
(define (fac n)
  (if (= n 0)
      1
      (* n (fac (- n 1)))))
(fac 5) ; => 120
```

Equivalent to …

RACKET

```
(letrec
 ([fac
   (λ (n)
    (if (= n 0)
        1
        (* n (fac (- n 1))))])
  (fac 5)) ; => 120
```

bind/rec examples

```
;; A Prog is one of:
;; …
;; - `(bind/rec [,Var ,Prog] ,Prog)
;; - `(iffy ,Prog ,Prog ,Prog)
;; …
```

JS "truthy if" (hw10)

RACKET

```
(letrec
 ([fac
   (λ (n)
    (if (= n 0)
        1
        (* n (fac (- n 1))))])
  (fac 5)) ; => 120
```

Equivalent to …

CS450LANG

```
(bind/rec
 [fac
  (lm (n)
    (iffy n
          (* n (fac (- n 1)))
          1))]
  (fac 5)) ; => 120
```

Zero is "falsy" (hw10)

Need new primitive in INIT-ENV

Racket define is lambda

Racket

```
(define (f n)
  (- n 1))
```

Equivalent to ...

Racket

```
(define f
  (λ (n)
    (- n 1)))
```

RACKET define is lambda and letrec

RACKET

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Equivalent to …

RACKET

```
(define factorial
  (letrec
   ([fac
     (λ (n)
      (if (= n 0)
          1
          (* n (fac (- n 1))))])
   fac)
```

In-class programming: map using letrec

RACKET

```
(define (map f lst)
  (if (null? lst)
      empty
      (cons (f (first lst)) (map f (rest lst)))))
```

Equivalent to …

RACKET

```
(define map
  (letrec
   ([_map
     (λ (n)
      (if (null? lst)
          empty
          (cons (f (first lst)) (_map f (rest lst)))))])
   _map)
```

Running **bind/rec** programs

```
;; A Prog is one of:
;; …
;; - `(bind/rec [,Var ,Prog] ,Prog)
;; …
```

parse

```
;; An AST is one of:
;; …
;; - (mk-recb Symbol AST AST)
;; …
(struct recb [var expr body])
```

run

```
;; A Result is a:
;; - …
```

Running **bind/rec** programs

TEMPLATE ?

```
;; run: AST -> Result
;; Computes result of
running CS450 Lang AST
```

```
;; An AST is one of:
;; …
;; - (mk-recb Symbol AST AST)
;; …
(struct recb [var expr body])
```

run

```
;; A Result is a:
;; - …
```

Running `bind/rec`

TEMPLATE : extract pieces

```
;; run: AST -> Result
(define (run p)

  (define (run/e p env)
    (match p
      …
     [(recb x e body) ?? x   ??      e       ??      body )))]
      … ))
 (run/e p  ???  ))
```

```
;; An AST is one of:
;; …
;; - (mk-recb Symbol AST AST)
;; …
(struct recb [var expr body])
```

Running bind/rec

TEMPLATE : recursive call

```
;; run: AST -> Result
(define (run p)

  (define (run/e p env)
    (match p
      …
     [(recb x e body) ?? x ?? (run/e e ??) ?? (run/e body ??) ))]
      … ))
 (run/e p  ???  ))
```

```
;; An AST is one of:
;; …
;; - (mk-recb Symbol AST AST)
;; …
(struct recb [var expr body])
```

Running **bind/rec**, using environment

```
;; run: AST -> Result
```

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

```
(define (run p)
  ;; accumulator env : Environment
  (define (run/e p env)
    (match p
      …
     [(recb x e body) ?? x ?? (run/e e ??) ?? (run/e body ??) ))]
      … ))
 (run/e p INIT-ENV ))
```

Running **bind/rec**, using environment

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  (define (run/e p env)
    (match p
      …
     [(recb x e body)
      (define env/x (env-add env x (run/e e env))
      (run/e body env/x)]
      … ))
 (run/e p INIT-ENV ))
```

2. add x binding to environment

1. Compute Result for x

Running **bind/rec**, using environment

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  (define (run/e p env)
    (match p
      …
     [(recb x e body)
      (define env/x (env-add env x (run/e e env/x
      (run/e body env/x)]
      … ))
 (run/e p INIT-ENV
```

CS450LANG

```
(bind/rec
 [fac
  (lm (n)
    (iffy n
          (* n (fac (- n 1)))
          1))]
  (fac 5)) ; => 120
```

??? This is circular! (no base case)

Compute body with x in-scope

PROBLEM:
x <u>should be in-scope here too</u>!

Interlude: Mutation

- **Mutating** a variable means: to <u>change</u> its value <u>after</u> it is defined

```
(define x 3)
(display x) ; 3
(set! x 5) ; mutate x
(display x) ; 5
```

Interlude: Mutation

- **Mutating** a variable means to change its value after it is defined
- **Mutation** should be <u>rarely used</u>, only in appropriate situations

Interlude: Mutation

- **Mutating** a variable means to change its value after it is defined

- **Mutation** should be <u>rarely used</u>, only in appropriate situations

> *Item 3: Use const whenever possible.*
> **Effective C++**, Scott Meyers, 2005.

Item 15, "Minimize mutability. **Joshua Bloch** Author, Effective Java, Second Edition

Joshua Bloch, Google's chief Java architect, is a former Distinguished Engineer at Sun Microsystems, where he led the design and implementation of numerous Java platform features, including JDK 5.0 language enhancements and the award-winning Java Collections Framework.

Immutability makes <u>code easier to read</u> and understand

Item 15 tells you to keep the state space of each object as simple as possible. If an object is immutable, it can be in only one state, and you win big. You never have to worry about what state the object is in, and you can share it freely, with no need for synchronization. If you can't make an object immutable, at least minimize the amount of mutation that is possible. This makes it easier to use the object correctly.

Interlude: Mutation

- **Mutating** a variable means to change its value after it is defined

- **Mutation** should be <u>rarely used</u>, only in appropriate situations

Because:

- It **makes code more difficult to read**
  - (just like inheritance and dynamic scope)
- It **violates "Separation of concerns"**

```
(define x 3)
(do-something x) ; mutate x??
(display x) ; ???
```

Interlude: Mutation

- **Mutating** a variable means to change its value after it is defined

- **Mutation** should be <u>rarely used</u>

When is using **mutation** ok:

- Performance
  - Typically **not using high-level languages!** (**OS, AAA game** i.e., not this class!)
  - Beware of **pre-mature optimization!**
- **Shared state** (in distributed programs)
  - Beware of **race conditions and deadlock!**
- **Circular data structures** (e.g., circular lists)

Running **bind/rec**, recursive environment items

```
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p     …
     [(recb x e body)

      (define env/x (env-add env x (run/e e env/x)))


      (run/env body env/x)]
                 …  ))
 (run/e p INIT-ENV ))
```

??? This is **circular**! (no base case)

PROBLEM:
x should be in-scope here too!

Compute body with x in-scope

Running **bind/rec**, recursive environment items

```
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p    …
     [(recb x e body)
      (define placeholder (box CIRCULAR-ERROR))
      (define env/x (env-add env x placeholder))



      (run/env body env/x)]
                 … ))
 (run/e p INIT-ENV ))
```

Creates **mutable box**
Makes mutation explicit

```
;; A Result is a:
;; - Number
;; - FunctionResult
;; - ErrorResult
```

```
;; An ErrorResult is a:
;; - UNDEFINED-ERROR
;; - ARITY-ERROR
;; - CIRCULAR-ERROR
```

Running **bind/rec**, recursive environment items

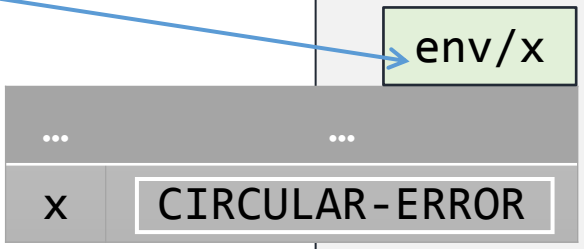
```
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p     …
     [(recb x e body)
      (define placeholder (box CIRCULAR-ERROR)
      (define env/x (env-add env x placeholder)


      (run/env body env/x)]
                 … ))
 (run/e p INIT-ENV ))
```

```
;; An Environment (OLD) (Env) is one of:
;; - empty                    ???
;; - (cons (list Var Result) Env)
```

(how would env-add and env-lookup need to change?)

```
;; An Environment is a: List<(list Var EnvVal)>
```

```
;; An EnvVal is one of:
;; - Result
;; - Box<Result>
```



Running **bind/rec**, recursive environment items

CS450LANG

```
(bind/rec [f f] f)
 ; => CIRCULAR-ERROR
```

```
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p     …
     [(recb x e body)
      (define placeholder (box CIRCULAR-ERROR))
      (define env/x (env-add env x placeholder))
      (define x-result (run/env e env/x))

      (run/env body env/x)]
                 … ))
 (run/e p INIT-ENV ))
```

Non-function, circular recursive references (no base case) produce error results!

Compute **x**'s `Result` with **x** in-scope!

env/x

…	…
x	CIRCULAR-ERROR

Running **bind/rec**, recursive environment items

```
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p    …
     [(recb x e body)
      (define placeholder (box CIRCULAR-ERROR)
      (define env/x (env-add env x placeholder)
      (define x-result (run/env e env/x)
      (set-box! placeholder x-result)
      (run/env body env/x)]
                 … ))
  (run/e p INIT-ENV ))
```

Close the (circular data structure) loop, with **mutation!**

Explicitly **mutate** mutable box

env/x

…	…
x	~~CIRCULAR-ERROR~~ **x-result**

Running **bind/rec**, recursive environment items

CS450LANG

```
(bind/rec
 [fac
  (lm (n)
    (iffy n
          (* n (fac (- n 1)))
          1))]
  (fac 5)) ; => 120
```

```
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p     …
     [(recb x e body)
      (define placeholder (box CIRCULAR-ERROR))
      (define env/x (env-add env x placeholder))
      (define x-result (run/env e env/x))
      (set-box! placeholder x-result)
      (run/env body env/x)]
                 …  ))
 (run/e p INIT-ENV ))
```

Compute body with **x** in-scope

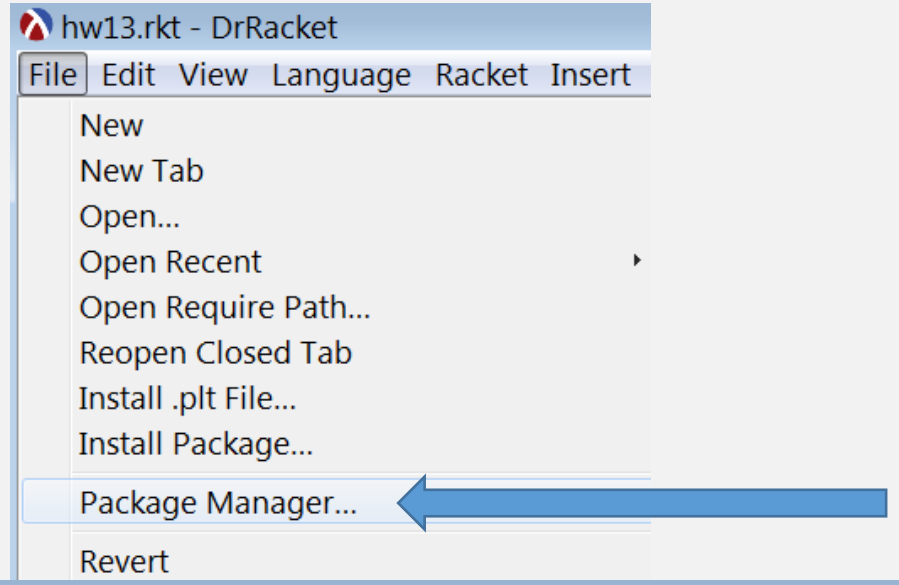
env/x

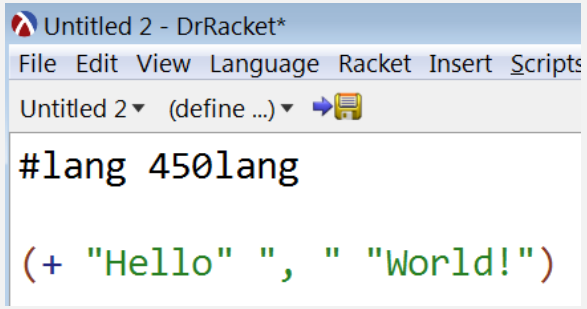
…	…
x	~~CIRCULAR-ERROR~~ **x-result**

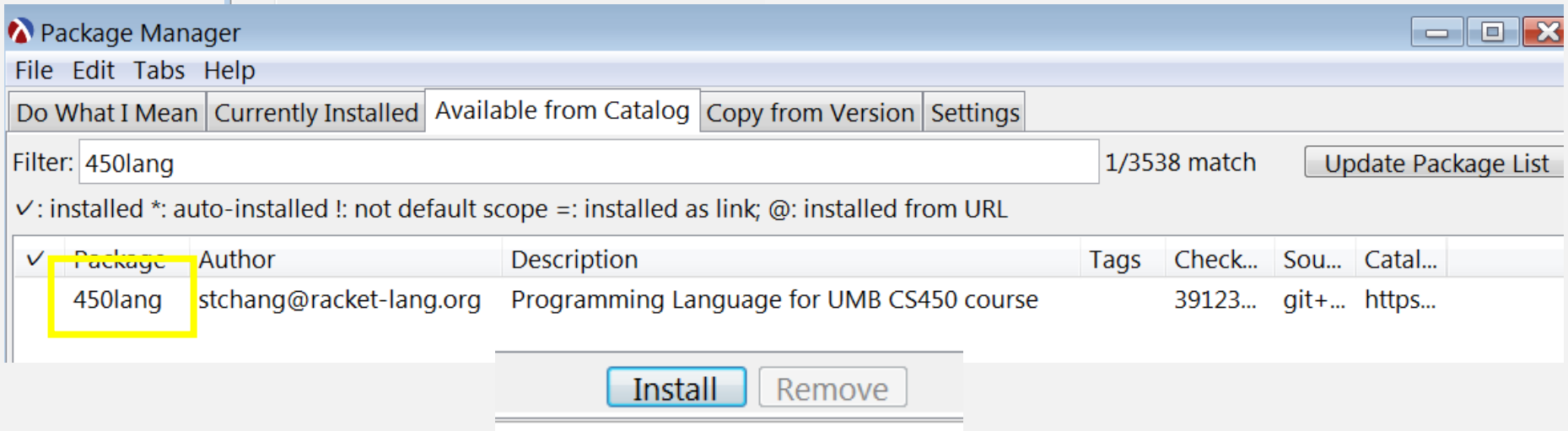
HW 13 Preview: Recursion!

Use “CS 450 LANG”! … to write (recursive) programs:

In-class: Install “450 Lang”







Extra credit: 450 Lang "bug bounty" (coming soon!)

- **Submit Bug reports that contain the following:**
  - An **informative** title that succinctly describes the problem,
  - a **minimal #LANG 450LANG** example that demonstrates the problem,
  - the expected result (with an explanation if necessary),
  - the current (incorrect) result.

Note: this means **you must figure out the correct expected behavior first!** (Your confusion is not always a bug in the software)

- **Total Possible Bonus: ??? points**

- This is a **real software project** with **real users** so **all submitted reports must meet real-world quality standards!**
  - Any submissions that do not follow instructions will be closed with no credit!
  - First come, first serve

In-class: Install “450 Lang”