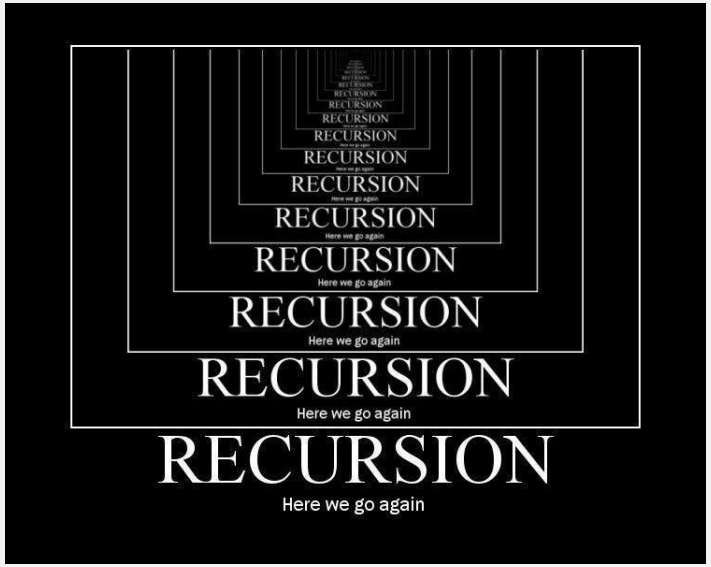
UMass Boston Computer Science

CS450 High Level Languages

Implementing Recursive Variables

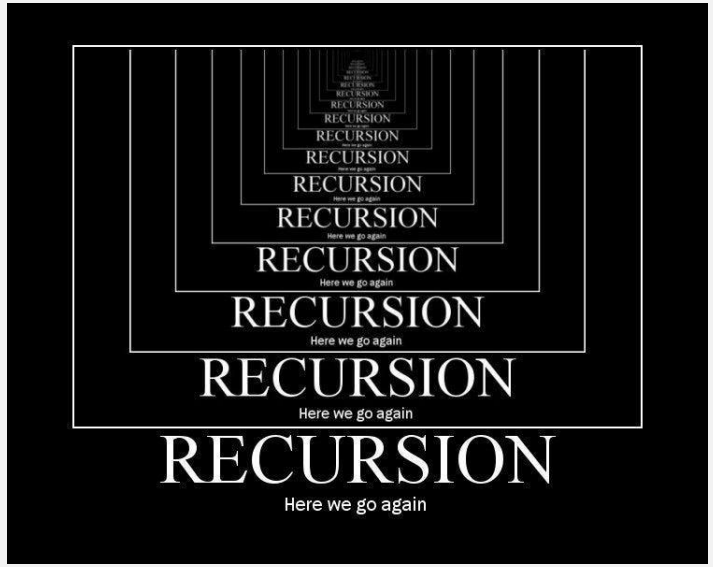
(with mutation??)

Thursday, April 23, 2026



Logistics

- HW 10 in
  - ~~due: Thu 4/23 11am EST~~
- HW 11 out
  - due: Tues 5/5 11am EST



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 a 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 )
```

???

`bind` examples, with functions

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

"lambda" function

function

arguments

function call

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

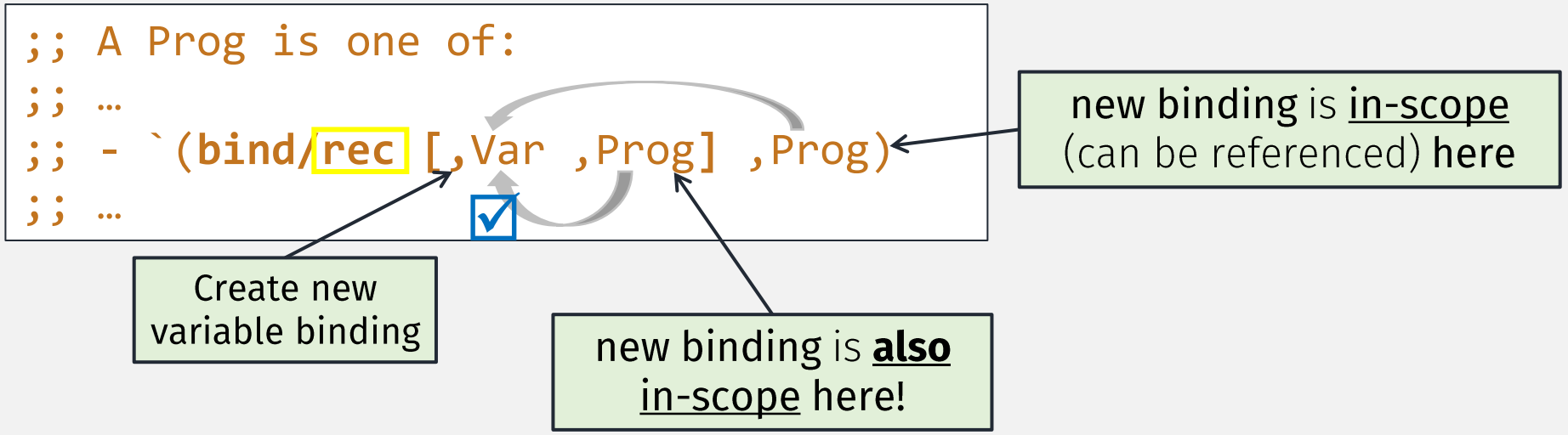
`f` not in-scope here
(so function can't be recursive!)

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

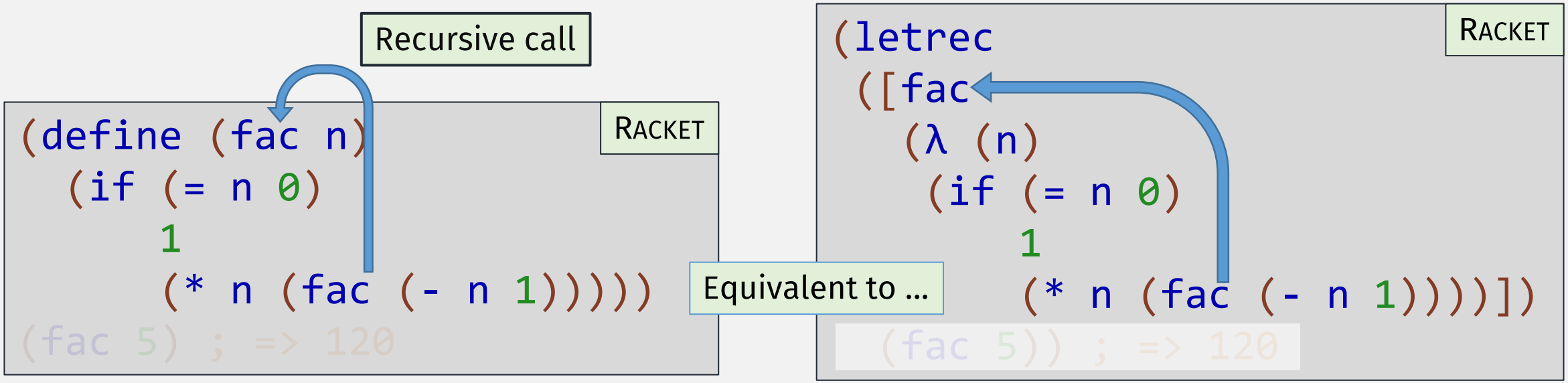
`f` not in-scope here
(so function can't be recursive!)

How to add recursion to our language?

`bind/rec` in “CS450” Lang



Racket recursive function examples



`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)

Assume “-” 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`

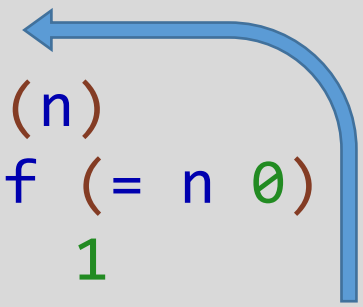
RACKET

```
(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: recursion with letrec

RACKET

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

Equivalent to …

Task: re-write your favorite recursive function using (non-function) **define** and **letrec**

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])
```

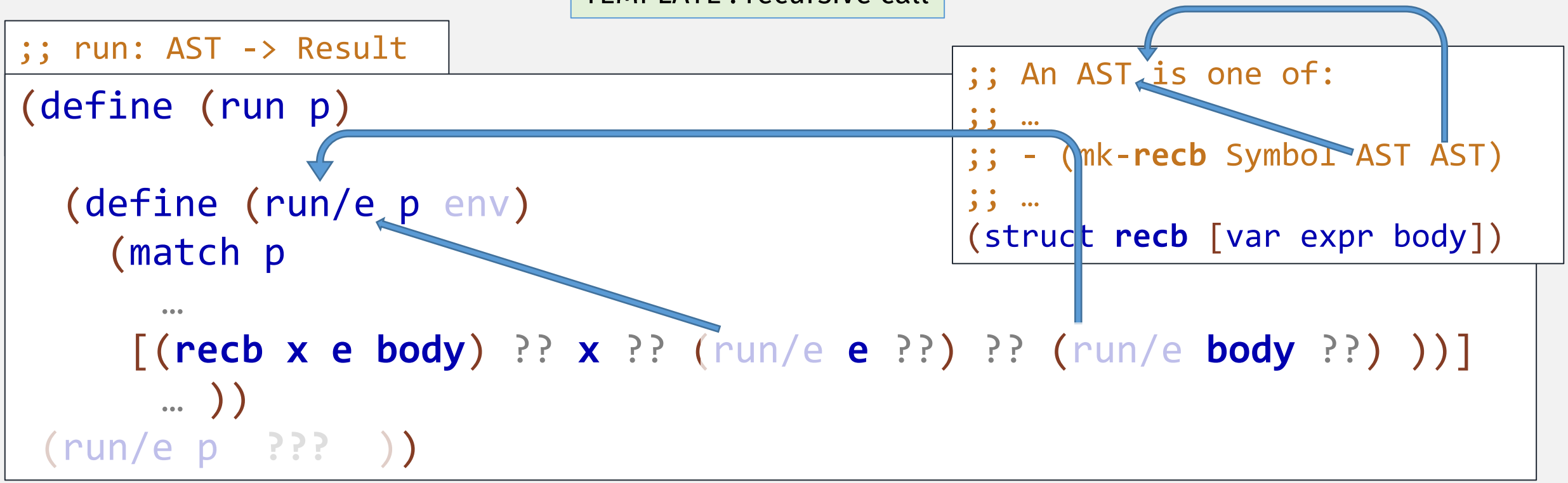
Running **bind/rec**

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
(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 ))
```

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) 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
       (run/e body env/x)]
      … ))
  (run/e p INIT-ENV
```

CS450LANG
Recursive Example

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

env/x

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

Compute body with x in-scope

PROBLEM:
x should be in-scope here too!

Interlude: Mutation

- **Mutating** a variable means: to change its value after 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 rarely used, only in appropriate situations

Interlude: Mutation

- **Mutating** a variable means: to change its value after it is defined
- **Mutation** should be rarely used, 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 **code** easier to read 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 rarely used, only in appropriate situations

Because:

- It **makes code more difficult to read** and **understand**
  - (just like other "bad" features, e.g., inheritance and dynamic scope)
- It **violates "Separation of concerns"**

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

Interlude: Mutation

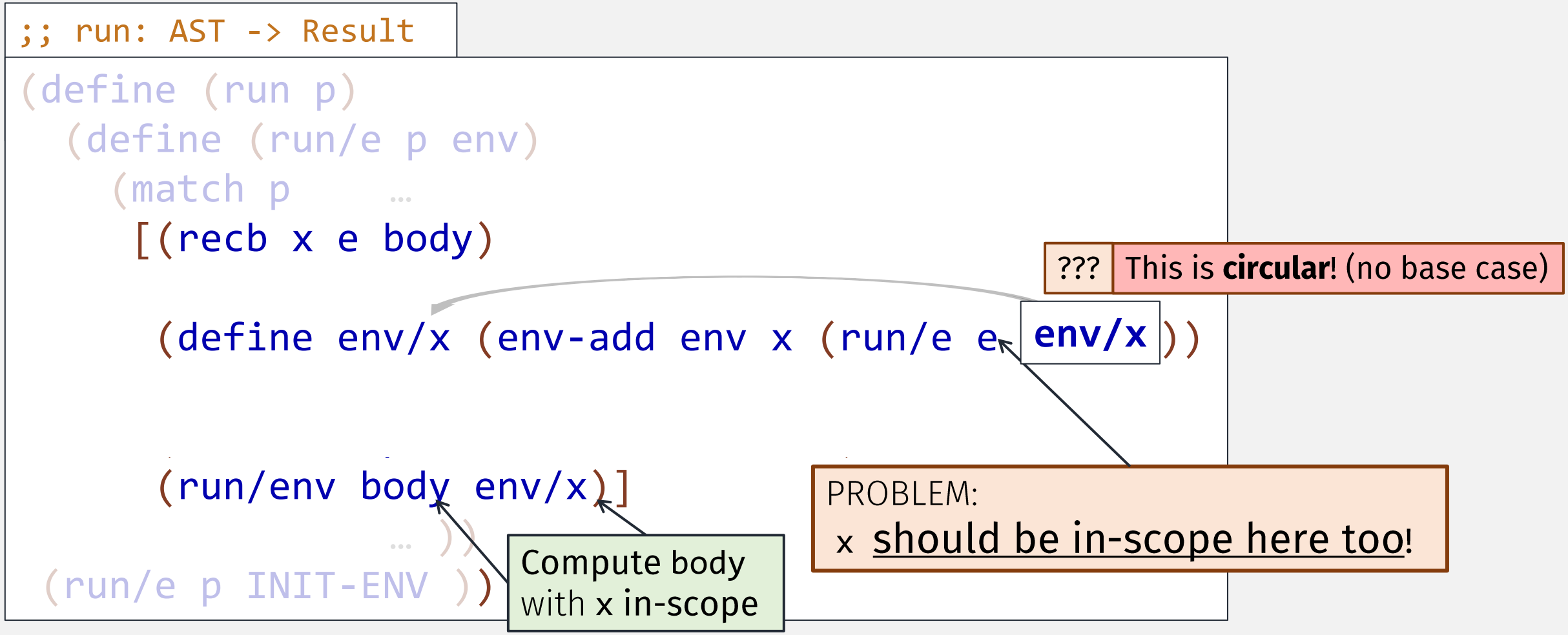
- **Mutating** a variable means: to change its value after it is defined

- **Mutation** should be rarely used

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** (don't optimize without hard profiling data)**!**
- **Shared state** (in distributed programs)
  - Beware of **race conditions and deadlock!**
- **Circular data structures** (e.g., circular lists)

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



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 ))
```

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

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

Creates **mutable box.**
Makes **mutation explicit**

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 ))
```

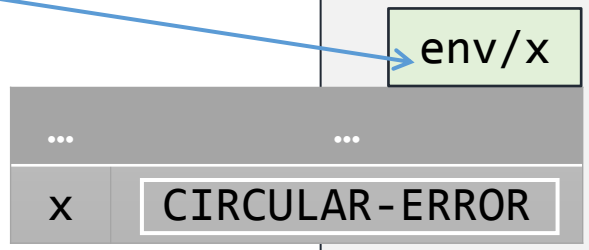
```
;; An Environment (OLD) (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

???

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

```
;; An Environment (Env) is a: Listof<(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

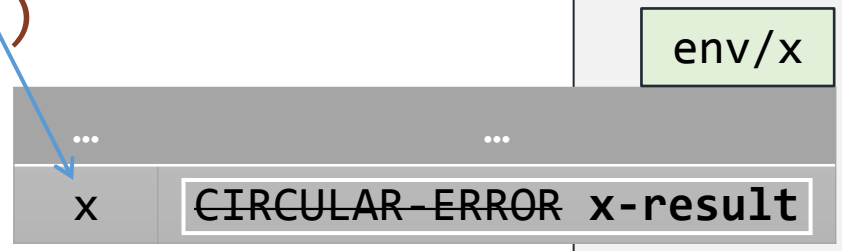
…	…
x	CIRCULAR-ERROR

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



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**