

UMass Boston Computer Science
CS450 High Level Languages

Implementing Recursion, Mutation

Thursday, May 1, 2025



Logistics

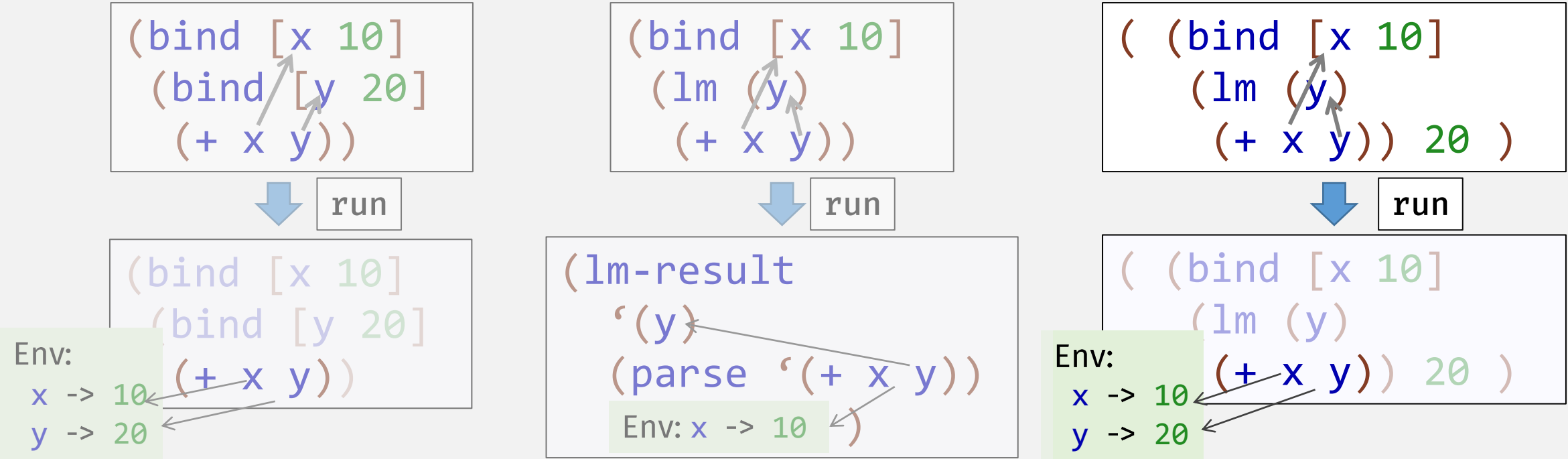
- HW 12 out
 - due: Tues 5/6 11am EST

(need “lambda” for hw12)

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



bind, lm, and their environments



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 )
```

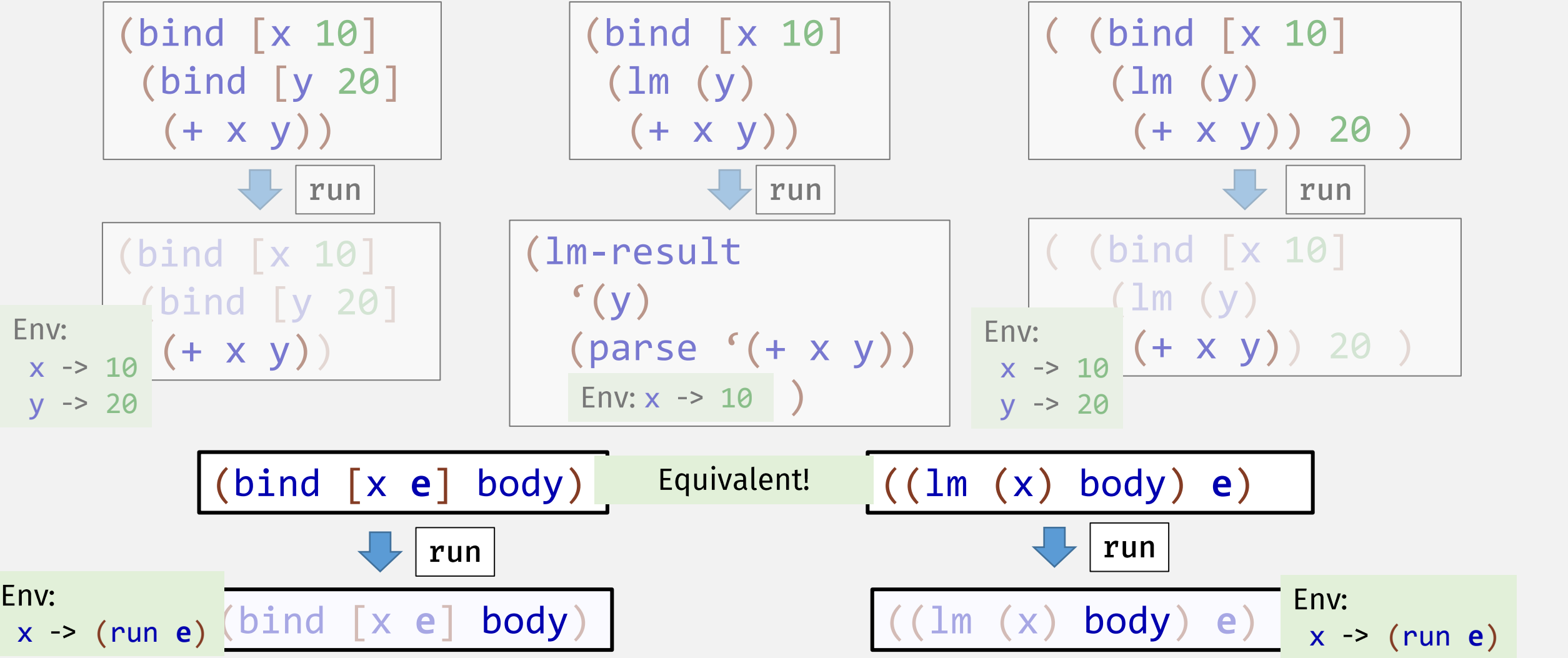
Env:

x -> 10

y -> 20

Last Time

bind = lm + fn call!



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>)  
;; ...
```

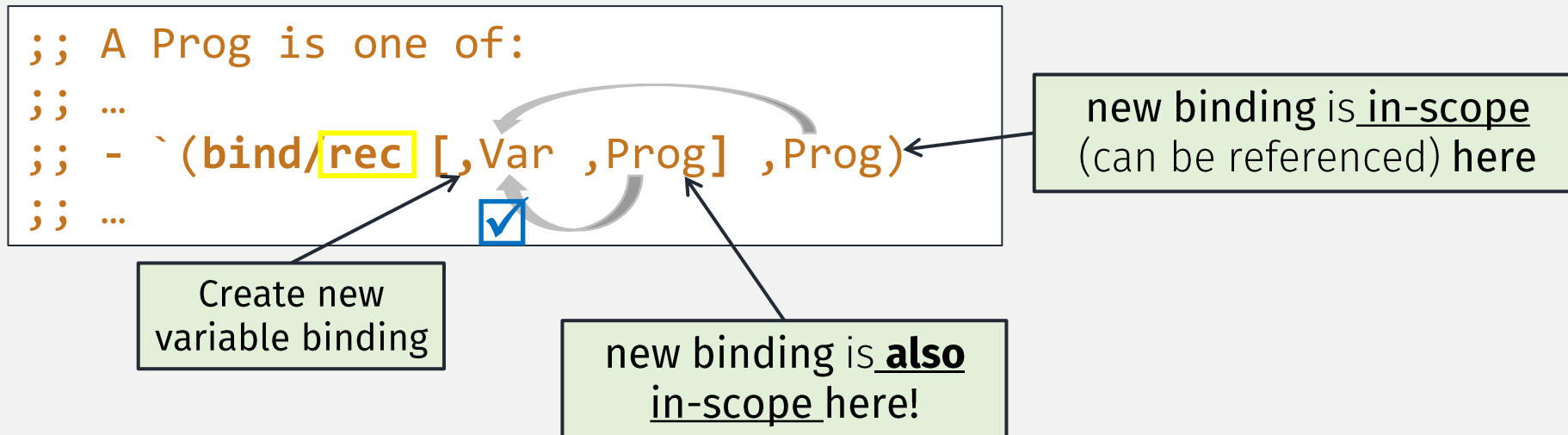
```
(check-equal?  
  (eval450  
    '(bind [f (lm (x) (+ x 4))]  
            (f 6)))  
    10 )
```

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

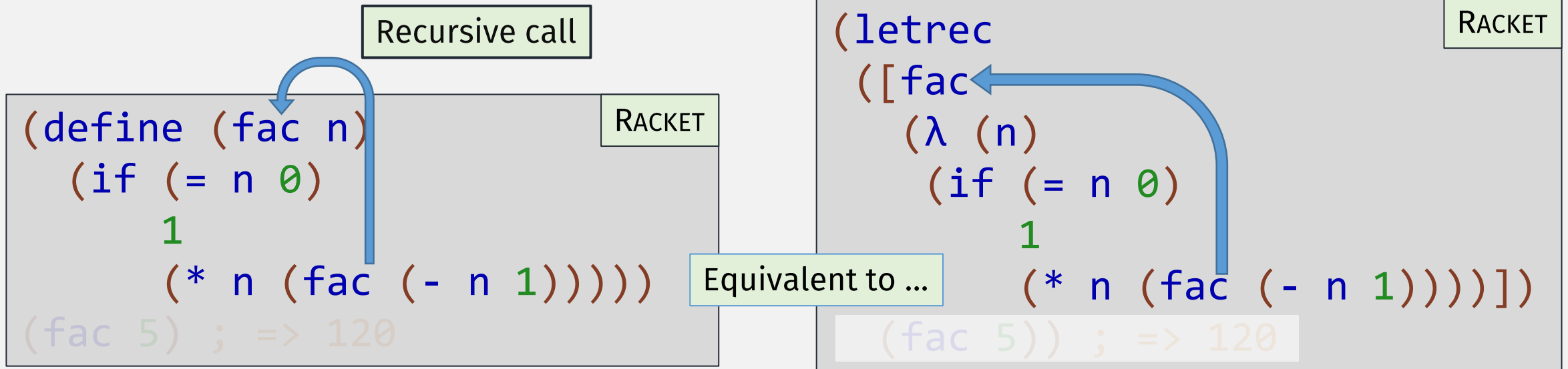
```
(check-equal?  
  (eval450  
    '(bind [f (lm (x) (f x))]  
            (f 6)))  
    UNDEF-ERR)
```

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

“bind/rec” in “CS450” Lang



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)

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

RACKET

Equivalent to ...

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

CS450LANG

Zero is “falsy” (hw10)

Need new
primitive in
INIT-ENV

RACKET define is lambda

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

RACKET

Equivalent to ...

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

RACKET

RACKET define is lambda and letrec

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

RACKET

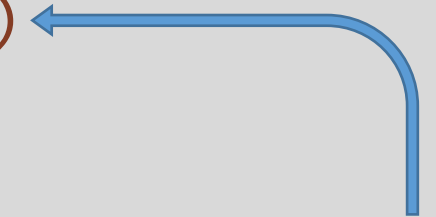
Equivalent to ...

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

RACKET

In-class programming: map using letrec

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

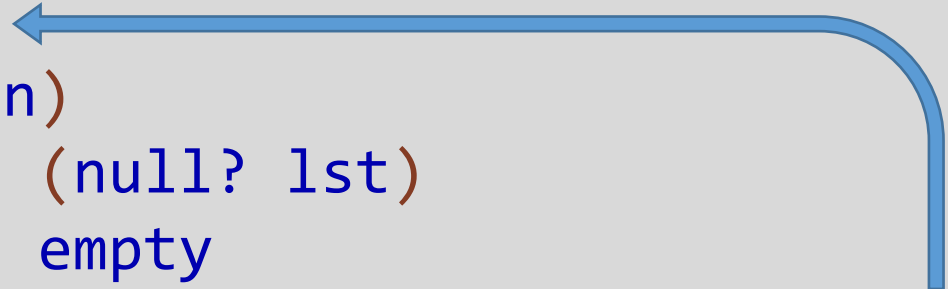


A blue arrow originates from the `(map f (rest lst))` expression in the recursive call and points back to the `(define (map f lst))` definition, illustrating the recursive nature of the function.

RACKET

Equivalent to ...

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



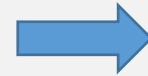
A blue arrow originates from the `(_map f (rest lst))` expression in the recursive call and points back to the `(letrec` definition, illustrating the recursive nature of the function.

RACKET

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 ??? ))
```

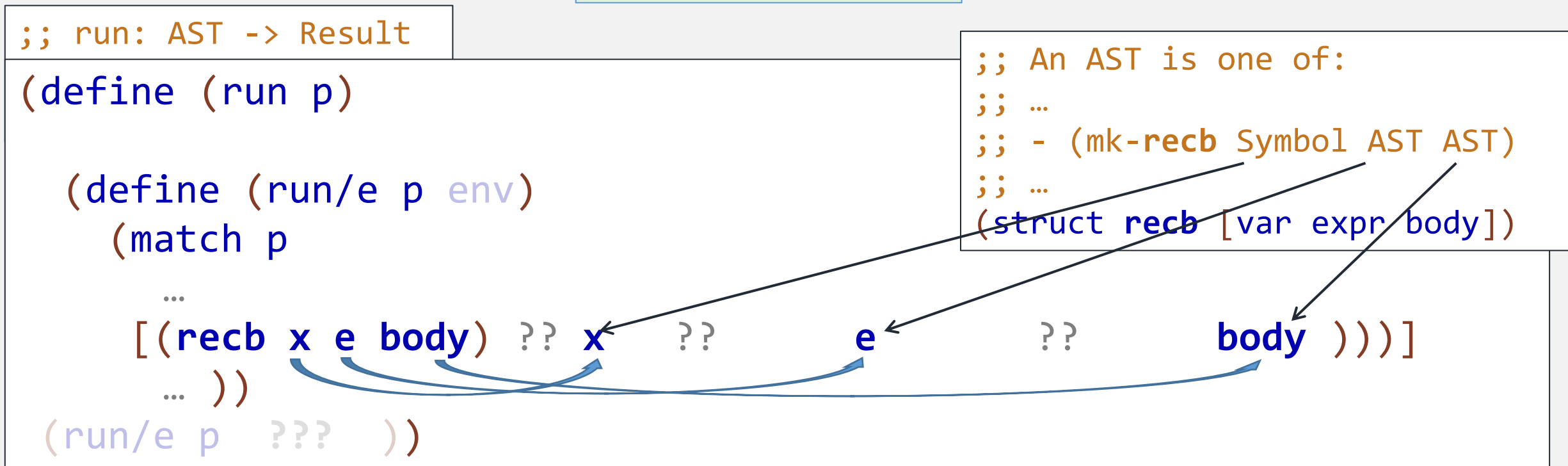
```
;; An AST is one of:
```

```
;; ...
```

```
;; - (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

CS450LANG

`;; 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))
```

Compute body
with x in-scope

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

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

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.

Immutability
makes code
easier to read
and understand

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.

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
 - (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 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!**
- **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>

env/x

| | |
|-----|----------------|
| ... | ... |
| x | CIRCULAR-ERROR |

Running `bind/rec`, recursive environment items

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

CS450LANG

;; 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



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 ))
```

Compute body
with **x** in-scope

CS450LANG

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



env/x



HW 13 Preview: Recursion!

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

In-class: Install “450 Lang”

The image shows a DrRacket environment with two windows. The top-left window, titled 'hw13.rkt - DrRacket', has its 'File' menu open, with 'Package Manager...' selected and highlighted by a blue arrow. The top-right window, titled 'Untitled 2 - DrRacket*', contains the following Racket code:

```
#lang 450lang

(+ "Hello" ", " "World!")
```

The bottom window is the 'Package Manager' dialog. It has tabs for 'Do What I Mean', 'Currently Installed', 'Available from Catalog', 'Copy from Version', and 'Settings'. The 'Available from Catalog' tab is active. The filter is set to '450lang', showing 1/3538 matches. Below the filter, a legend explains the symbols: '✓: installed', '*: auto-installed', '!: not default scope', '=: installed as link', and '@: installed from URL'. A table lists available packages, with '450lang' highlighted by a yellow box:

| ✓ | Package | Author | Description | Tags | Check... | Sou... | Catal... |
|---|---------|-------------------------|---|------|----------|---------|----------|
| ✓ | 450lang | stchang@racket-lang.org | Programming Language for UMB CS450 course | | 39123... | git+... | https... |

At the bottom of the Package Manager window are 'Install' and 'Remove' buttons.

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”

The screenshot illustrates the process of installing the '450lang' package in DrRacket. It shows three windows:

- hw13.rkt - DrRacket**: The main editor window with the 'File' menu open. The 'Package Manager...' option is highlighted, and a blue arrow points to it.
- Untitled 2 - DrRacket***: A new window showing the code for the '450lang' package:

```
#lang 450lang

(+ "Hello" ", " "World!")
```
- Package Manager**: A window showing the search results for '450lang'. The '450lang' package is listed with a checkmark in the 'Package' column, indicating it is installed. The 'Install' button is highlighted.

The Package Manager window displays the following information:

| Package | Author | Description | Tags | Check... | Sou... | Catal... |
|---------|-------------------------|---|------|----------|---------|----------|
| 450lang | stchang@racket-lang.org | Programming Language for UMB CS450 course | | 39123... | git+... | https... |

Buttons at the bottom of the Package Manager window: **Install** and **Remove**.