UMass Boston Computer Science

CS450 High Level Languages (section 2)

Interpreting Recursion, with Mutation!

Monday, December 2, 2024

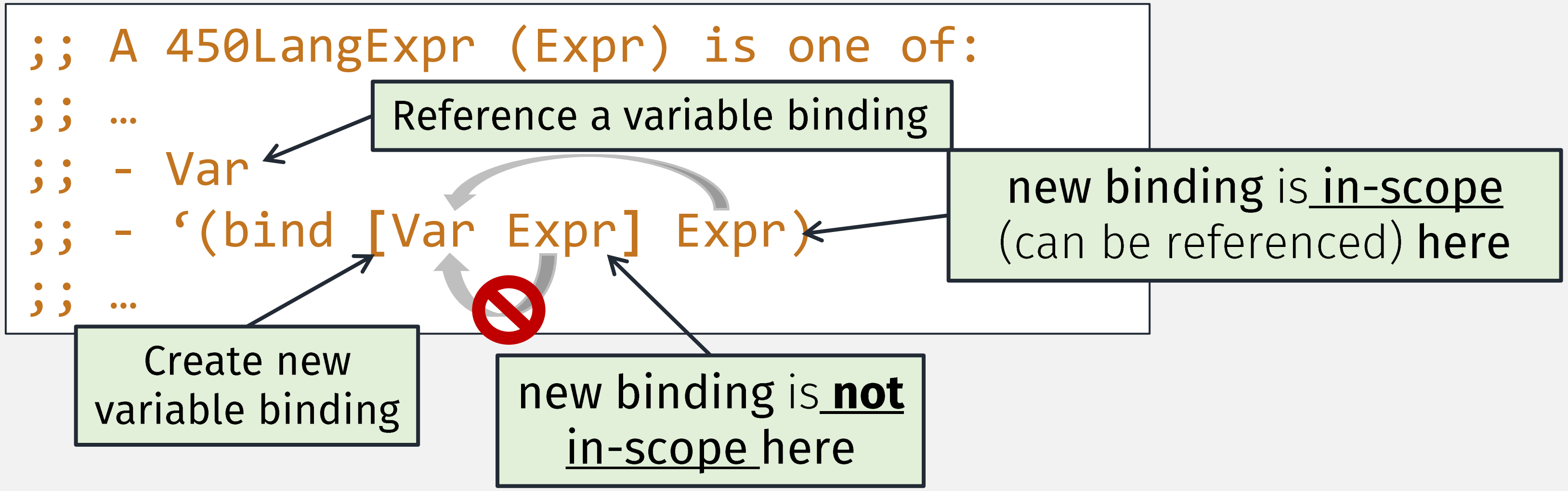
Logistics

- HW 12 out
  - due: Sun 12/4 12pm (noon) EST

Last Time

“bind” in “CS450” Lang

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



```
;; A 450LangExpr (Expr) is one of:
;; …
;; - Var
;; - ‘(bind [Var Expr] Expr)
;; …
```

Reference a variable binding

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

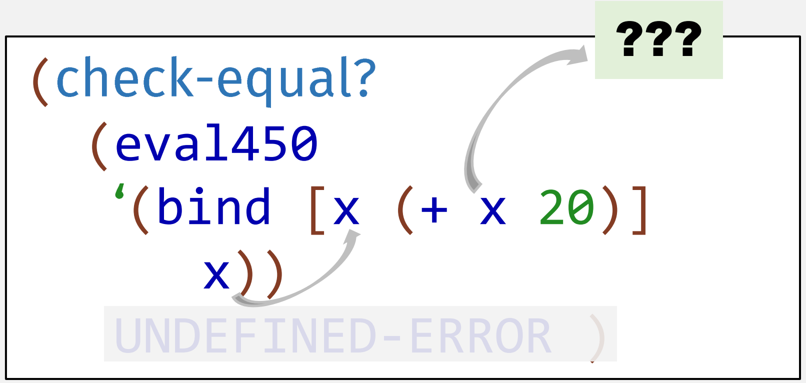
Create new variable binding

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

Last Time

bind examples

```
;; A 450LangExpr (Expr) is one of:
;; …
;; - Var
;; - ‘(bind [Var Expr] Expr)
;; …
```



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

bind examples, with functions

```
;; A 450LangExpr (Expr) is one of:
;; …
;; - Var
;; - ‘(bind [Var Expr] Expr)
     ‘(fn List<Var> Expr)
     (cons Expr List<Expr>)
;; …
```

“lambda” function

function

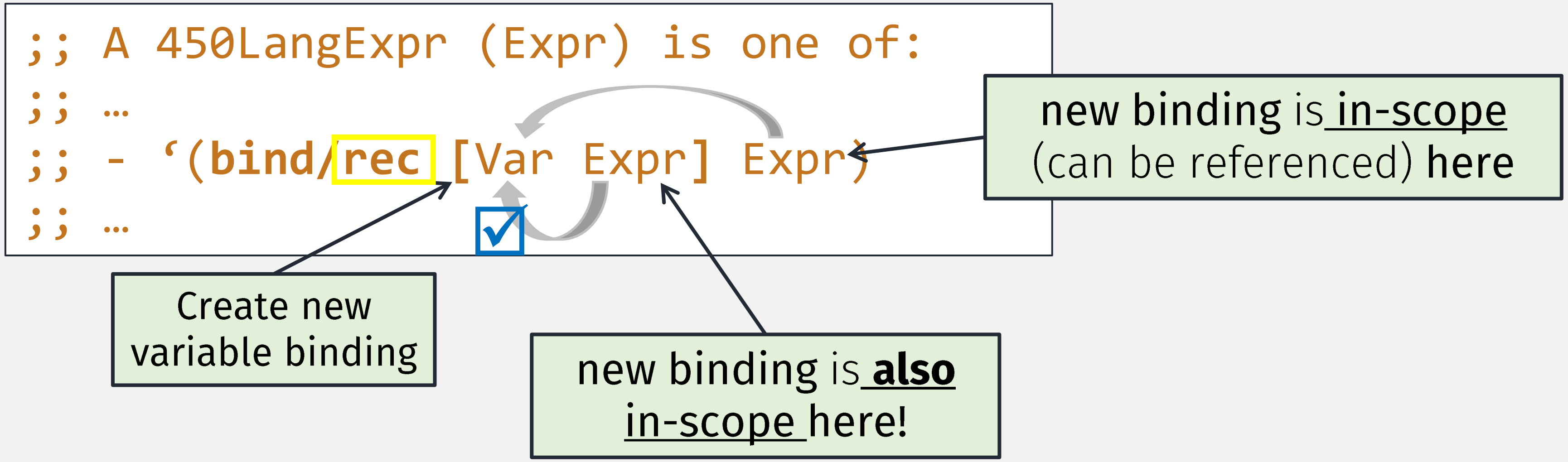
arguments

function call

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

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

“bind/rec” in “CS450” Lang



```
;; A 450LangExpr (Expr) is one of:
;; …
;; - ‘(bind/rec [Var Expr] Expr)
;; …
```

☑

Create new variable binding

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

new binding is in-scope (can be referenced) **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 450LangExpr (Expr) is one of:
;; …
;; - ‘(bind/rec [Var Expr] Expr)
;; - ‘(Expr ? Expr : Expr)
;; …
```

JS “truthy if” (hw10)

RACKET

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

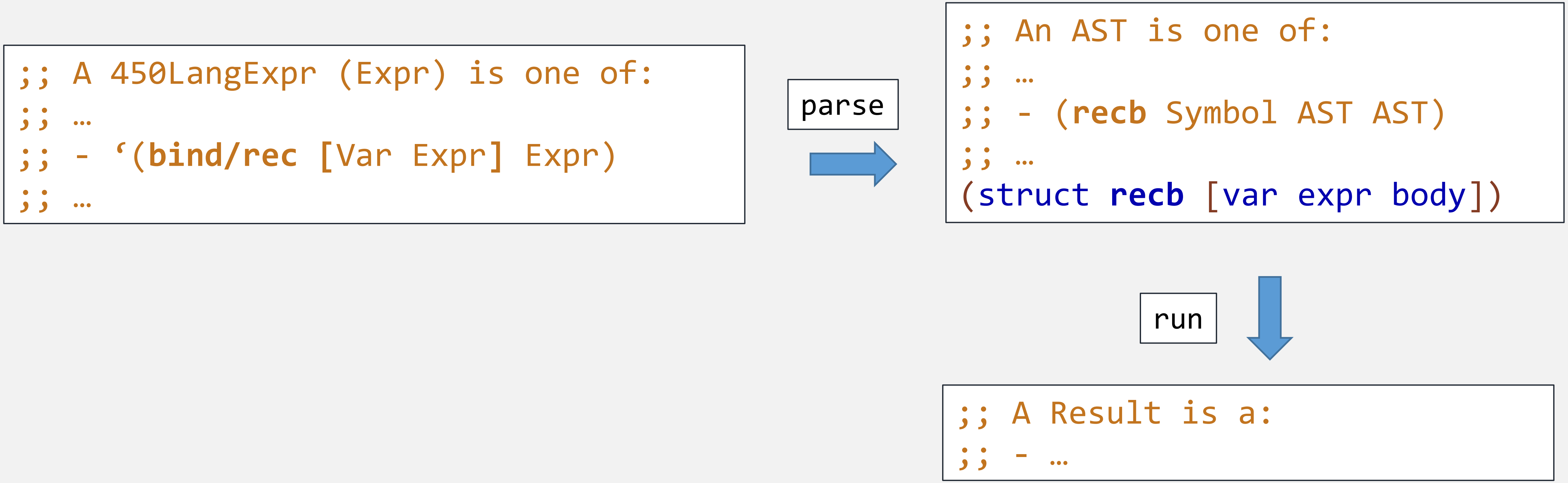
Equivalent to …

CS450LANG

```
(bind/rec
 [fac
  (fn (n)
    (n ? (* n (fac (- n 1)))
       : 1))]
  (fac 5)) ; => 120
```

Zero is “truthy” false (hw10)

Running **bind/rec** programs



```
;; A 450LangExpr (Expr) is one of:
;; …
;; - ‘(bind/rec [Var Expr] Expr)
;; …
```

parse

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

run

```
;; A Result is a:
;; - …
```

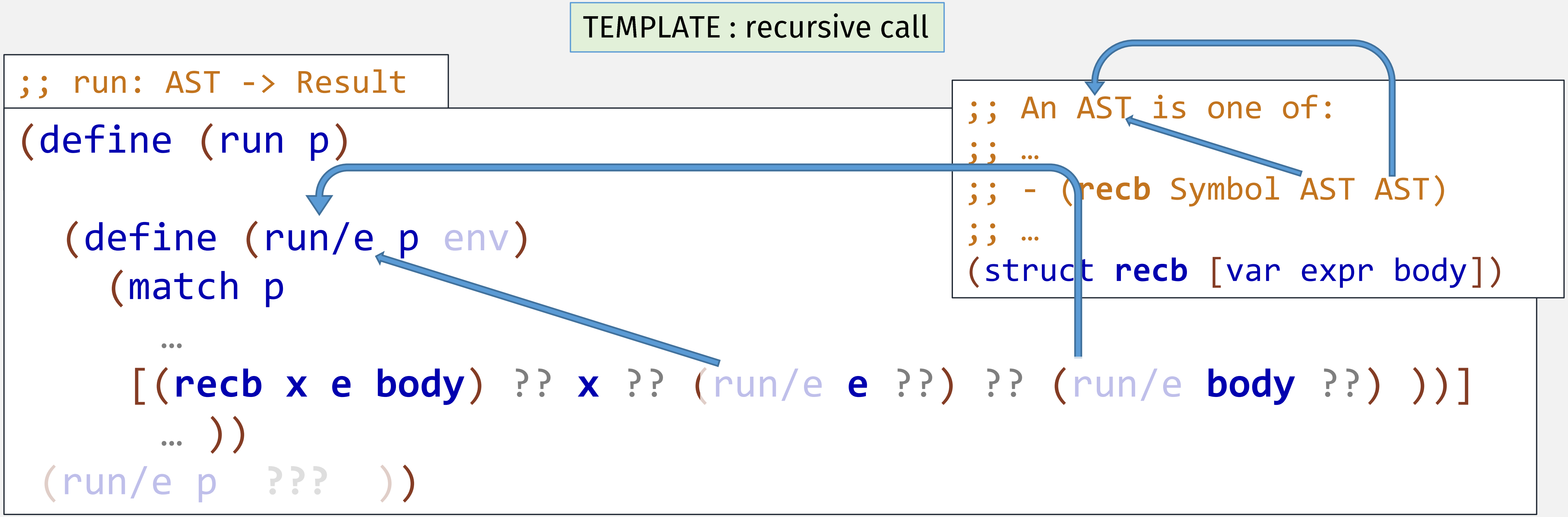
Running bind/rec

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:
;; …
;; - (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:
;; …
;; - (recb Symbol AST AST)
;; …
(struct recb [var expr body])
```

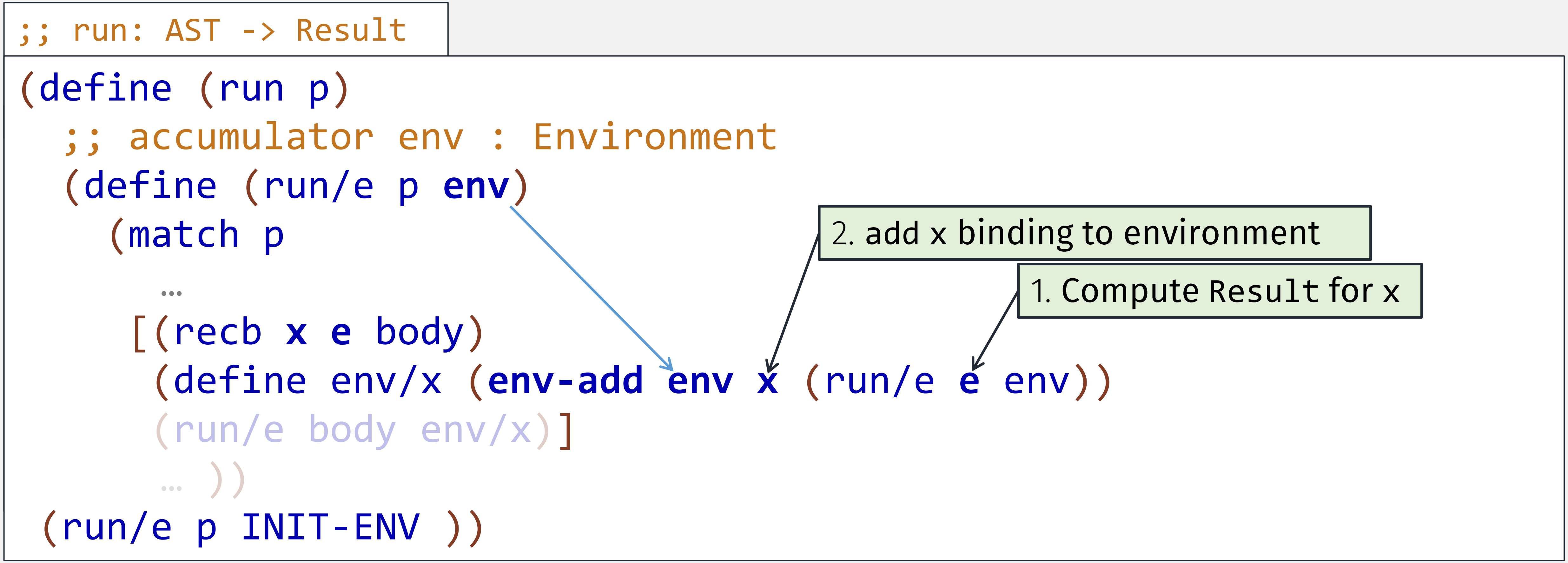
Running **bind/rec**, using environment

```
;; 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

```
(bind/rec
 [fac
  (fn (n)
    (n ? (* n (fac (- n 1)))
       : 1))]
  (fac 5)) ; => 120
```

```
;; 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
```

??? 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

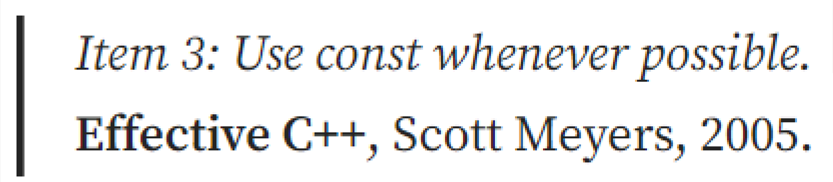
- **Mutating** a variable means to change its value after it is defined

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

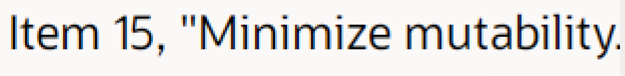
Interlude: Mutation

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

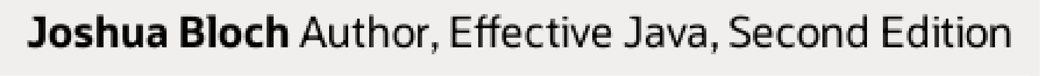
- **Mutation should be rarely used,** only in appropriate situations



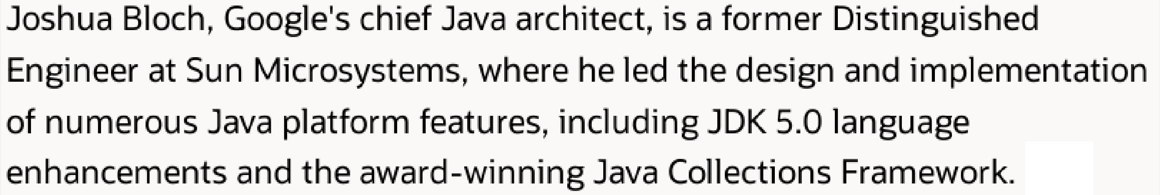
> *Item 3: Use const whenever possible.*
>
> **Effective C++**, Scott Meyers, 2005.



Item 15, "Minimize mutability.

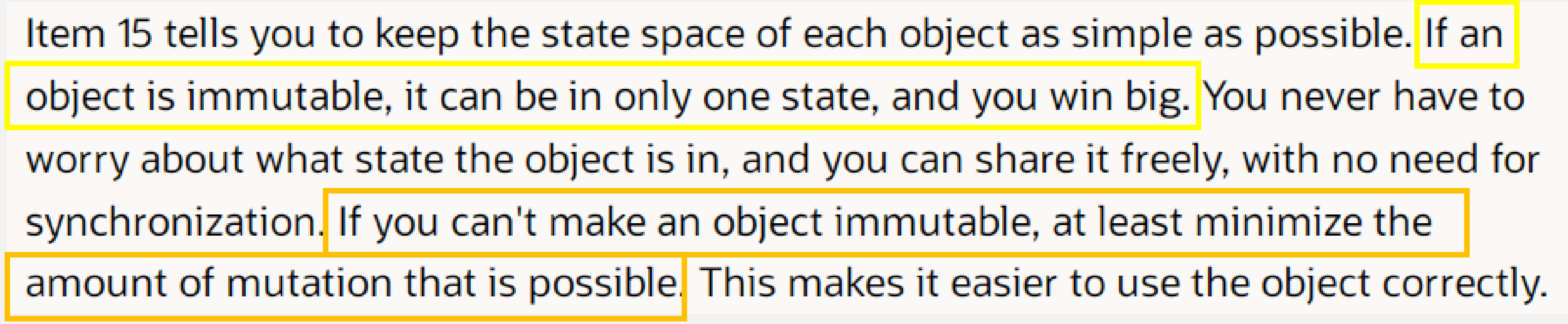


Joshua Bloch Author, Effective Java, Second Edition



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**
  - (just like inheritance and dynamic scope)
- It **violates "Separation of concerns"**

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

Interlude: Mutation

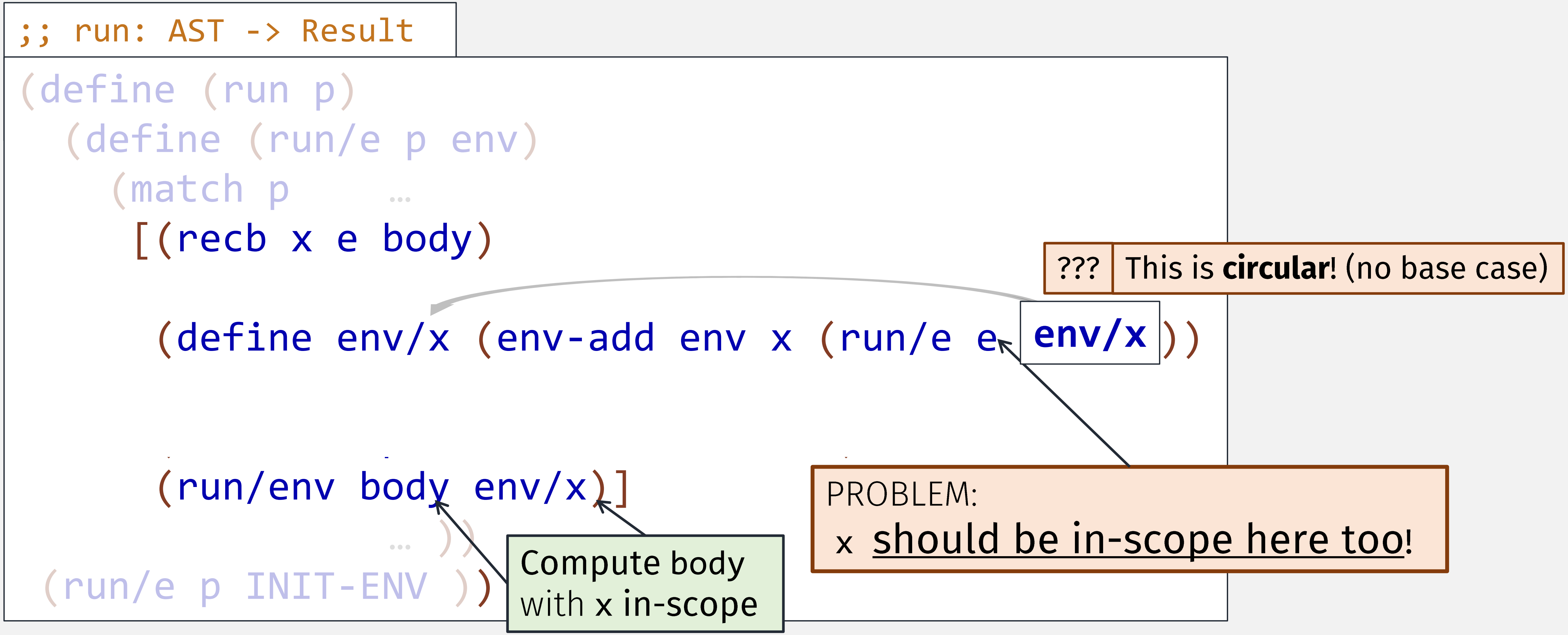
- **Mutating** a variable means to change its value after it is defined

- **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

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

```
;; 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

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

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

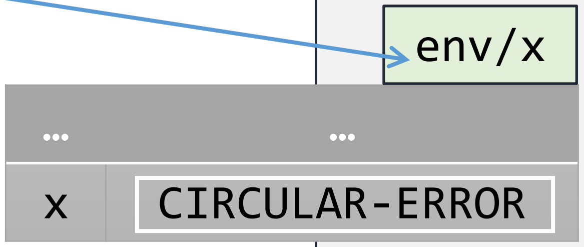
```
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p      …
     [(recb x e body)
      (define placeholder (box CIRCULAR-ERROR))
      (define env/x (env-add env x placeholder))


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

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

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

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

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



env/x

…	…
x	CIRCULAR-ERROR

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

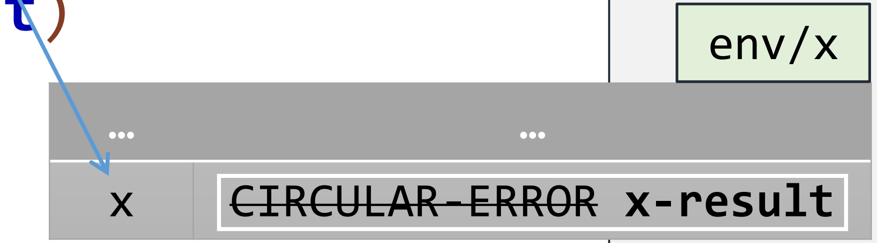
…	…
x	CIRCULAR-ERROR

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

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

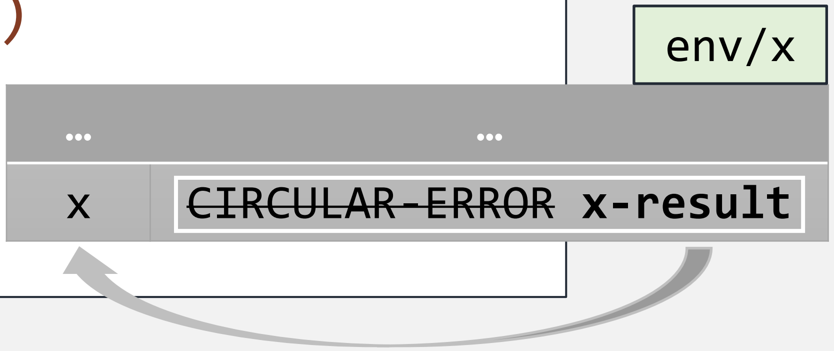
```
(define (run p)
  (define (run/e p env)
    (match p    …
     [(recb x e body)
      (define placeholder (box CIRCULAR-ERROR))
      (define env/x (env-add env x placeholder))
      (define x-result (run/env e env/x))
      (set-box! placeholder x-result)
      (run/env body env/x)]
           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

CS450Lang

```
(bind/rec
 [fac
  (fn (n)
    (n ? (* n (fac (- n 1)))
       : 1))]
  (fac 5)) ; => 120
```

☑

Compute body with **x** in-scope



env/x

…	…
x	~~CIRCULAR-ERROR~~ **x-result**

HW 13 Preview: Recursion!

Use “CS450 Lang”! … to write recursive programs:

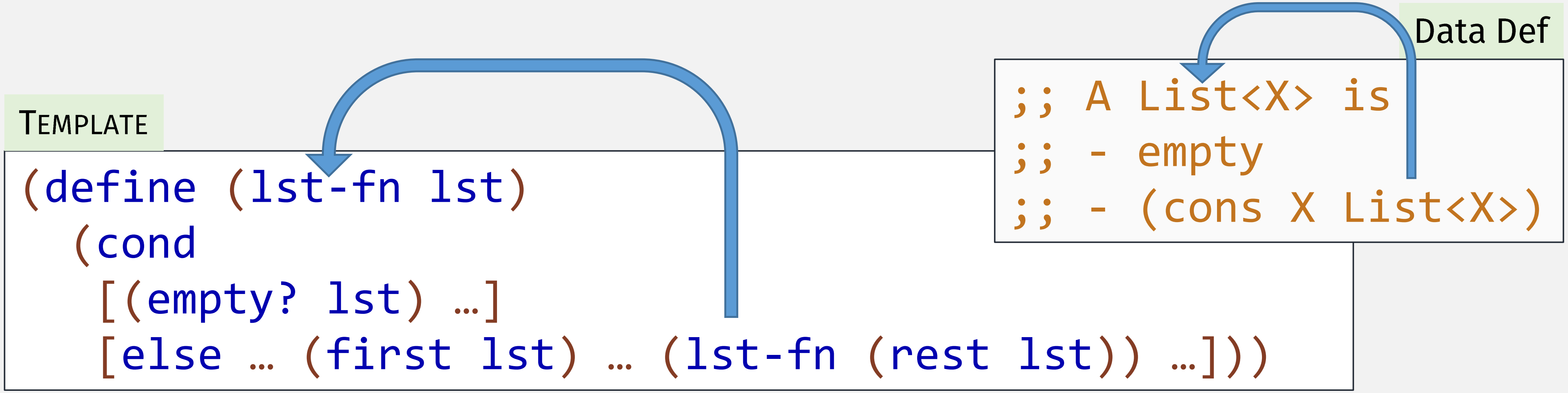
- `fac` (factorial)
- `filt` (filter)
- `qsort` (functional quicksort)
- `gcd`
- `sierpinski` (fractal)

(Extra primitives will be added to `INIT-ENV`, ask if you need more)

- Look it up if you don’t know any of these
  - Using any resources, e.g., ChatGPT, Co-pilot, is allowed
  - (still can’t submit else’s hw, obv)

Recursion review

- <u>Most</u> **recursion is structural** (comes from **data definitions**)!



TEMPLATE

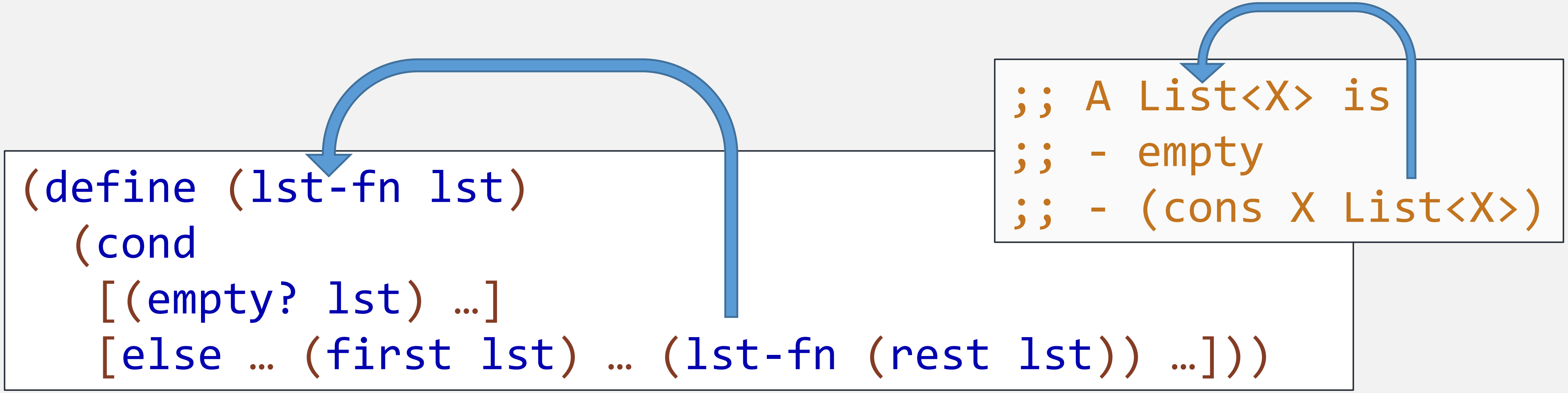
```
(define (lst-fn lst)
  (cond
    [(empty? lst) …]
    [else … (first lst) … (lst-fn (rest lst)) …]))
```

Data Def

```
;; A List<X> is
;; - empty
;; - (cons X List<X>)
```

A Different Kind of Recursion!

- Not all **recursion is structural** (comes from **data definitions**)!



```
(define (lst-fn lst)
  (cond
   [(empty? lst) …]
   [else … (first lst) … (lst-fn (rest lst)) …]))
```

```
;; A List<X> is
;; - empty
;; - (cons X List<X>)
```

A Different Kind of Recursion!

- **Not all recursion is structural** (comes from **data definitions**)!

```
;; gcd : Nat Nat -> Nat
;; computes greatest common divisor, using Euclid’s algorithm
;; termination argument:
;; m is halved (at least) eve                          y modulo fn)
(define (gcd n m)
  (if (= m 0)
      n
      (gcd m (modulo n m))
```

What template is this following??

A Different Kind of Recursion!

- **Non-structural recursion** (doesn’t come from **data definitions**) is called **generative recursion**
- **no template**, but requires **Termination Argument**
  - Explains why the function terminates!

```
;; gcd : Nat Nat -> Nat
;; computes greatest common divisor, using Euclid’s algorithm
;; termination argument:
;; m is halved (at least) every iteration (by modulo fn)
(define (gcd n m)
  (if (= m 0)
      n
      (gcd m (modulo n m))
```

Recursive call must be on “smaller” version of the problem