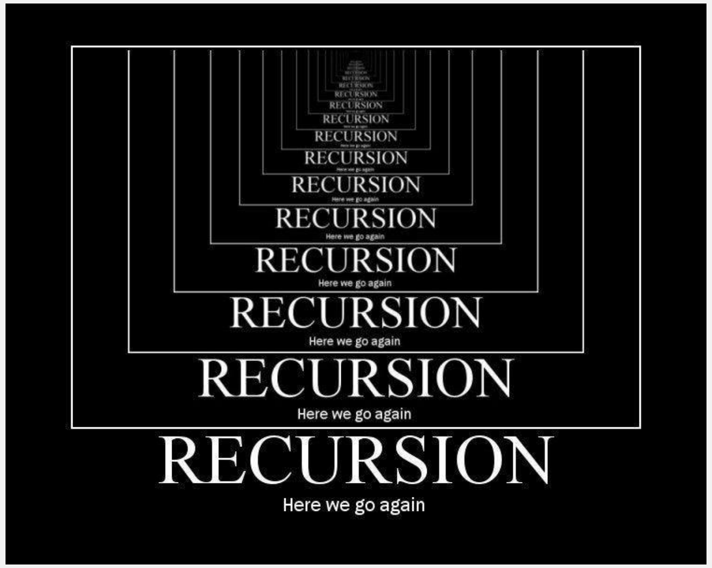
UMass Boston Computer Science

CS450 High Level Languages

Recursive Variables

Tuesday, April 29, 2025



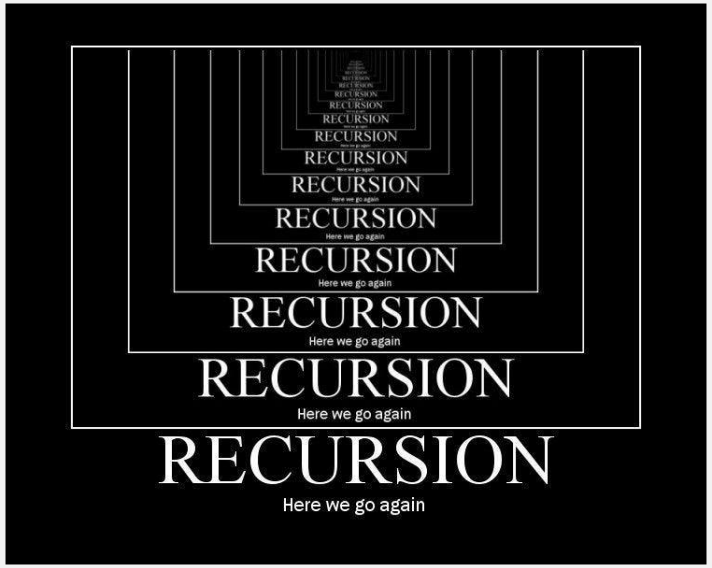
Logistics

- HW 11 in
  - ~~due: Tues 4/29 11am EST~~

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

    (need "lambda" for hw12)

    (don't need "recursive bind" for hw12)



Last Time

Function Application in CS450 Lang

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - `(bind [,Var ,Program] ,Program)
;; - (cons Program List<Program>)
```

function

arguments

What functions can be called?

```
(+ 1 2)
```

1. (Racket) functions in initial environment

```
(??? 1 2)
```

2. user-defined ("lambda") functions?

Last Time

“Lambdas” in CS450 Lang

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

Last Time

CS450 Lang "Lambda" examples

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

CS450LANG

```
(lm (x y) (+ x y))
```

Equivalent to ...

RACKET

```
(lambda (x y) (+ x y))
```

```
(lm (x) (lm (y) (+ x y)) ; "curried"
```

```
( (lm (x y) (+ x y)) 10 20 ) ; lm applied
```

Parsing "Lambda"

CS450LANG

```
(lm (x y) (+ x y))
```

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

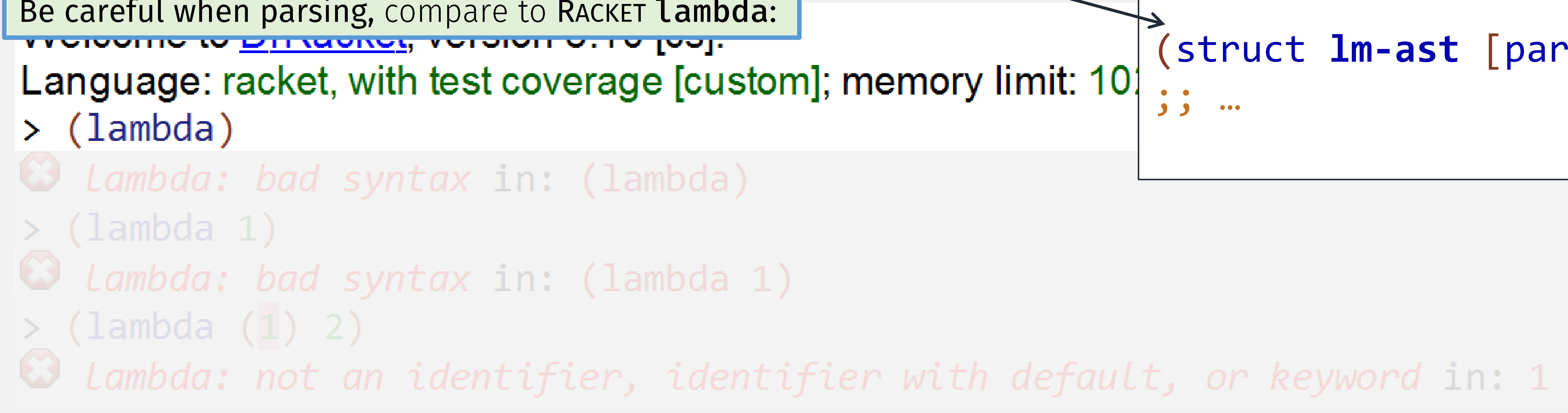
parse

```
;; An AST is one of:
;; …
;; - (mk-lm-ast List<Symbol> AST)
;; …



(struct lm-ast [params body])
;; …
```

Be careful when parsing, compare to RACKET **lambda**:



```
Welcome to DrRacket, version 8.10 [cs].
Language: racket, with test coverage [custom]; memory limit: 10
> (lambda)
lambda: bad syntax in: (lambda)
> (lambda 1)
lambda: bad syntax in: (lambda 1)
> (lambda (1) 2)
lambda: not an identifier, identifier with default, or keyword in: 1
```

Parsing “Lambda”

CS450LANG

```
(lm (x y) (+ x y))
```

```
(define/contract (parse p)
  (-> Program? AST?)
  (match p
     ...
   [`(lm ,(and (list (? symbol?) ...) args) ,bod) ... ]



   [`(,fn . ,args) ... ]
   [_ (raise-syntax-error
        'parse "not a valid CS450 Lang program" p
        #:exn exn:fail:syntax:cs450)]))
```

Correct syntax

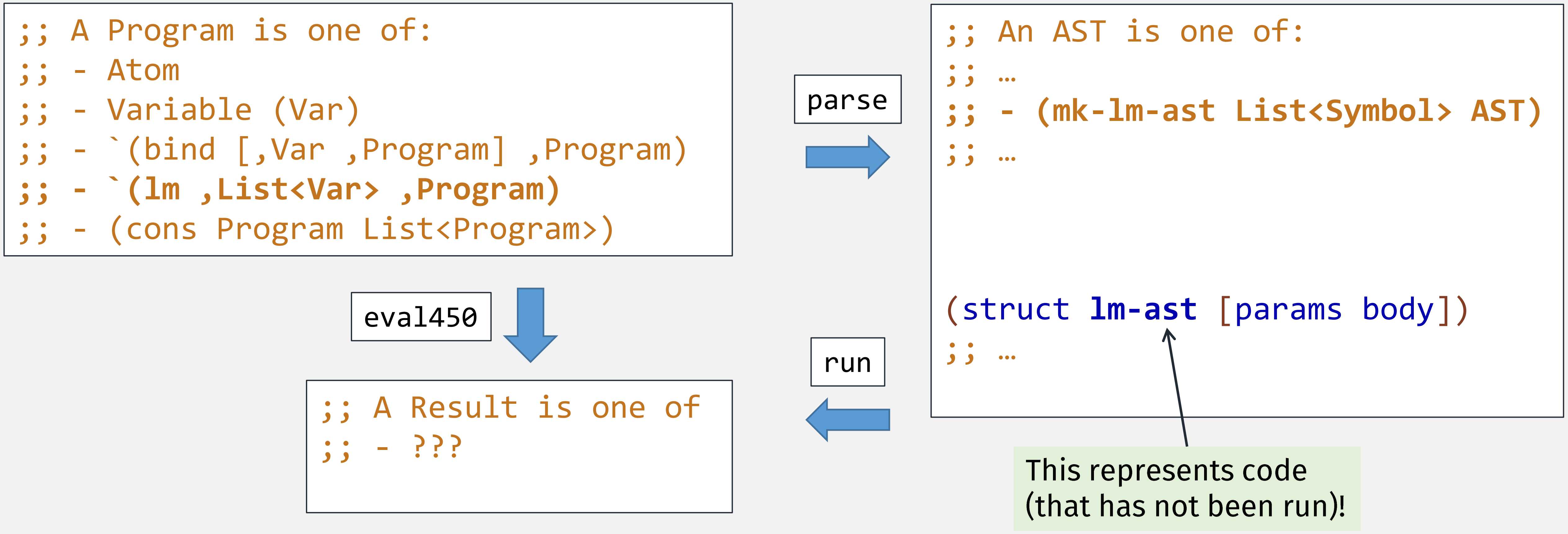
Parsing “Lambda”

```
(define/contract (parse p)
  (-> Program? AST?)
  (match p
      ...
   [`(lm ,(and (list (? symbol?) ...) args) ,bod) ... ]
   [`(lm . ,_)
    (raise-syntax-error 'parse “invalid lm syntax" p
      #:exn exn:fail:syntax:cs450)    ]
   [`(,fn . ,args) ... ]
   [_ (raise-syntax-error
      'parse "not a valid CS450 Lang program" p
      #:exn exn:fail:syntax:cs450)]))
```

“Lambda” parse error case

User-defined exception

CS450 Lang "Lambda" AST node



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

```
;; An AST is one of:
;; …
;; - (mk-lm-ast List<Symbol> AST)
;; …

(struct lm-ast [params body])
;; …
```

This represents code (that has not been run)!

```
;; A Result is one of
;; - ???
```

CS450 Lang “Lambda” full examples

```
(check-equal?
  (eval450
   ‘(bind [x 10]
     ( (lm (y) (+ x y)) 20 )))
   30 ) ; with bind
```

Expression that evaluates to a function result

```
(check-equal?
  (eval450
   ‘( (bind [x 10]
        (lm (y) (+ x y)))
      20 ))
   30 ) ; with bind (lm only)
```

argument

```
(check-equal?
  (eval450
   ‘( (lm (x y) (+ x y))
       10 20 ) )
   ? )
```

“Running” Functions?

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

```
(define (run p)


  (define (run/e p env)
    (match p
         …
     [(lm-ast params body) ??           ??                    ??]
         …
     ))
 (run/e p INIT-ENV))
```

TEMPLATE?

```
;; An AST is one of:
;; …
;; - (mk-lm-ast List<Symbol> AST)
;; …
(struct lm-ast [params body])
```

“Running” Functions?

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

  (define (run/e p env)
    (match p
        …
     [(lm-ast params body) ?? params ?? (run/e body env) ??]
        …
     ))
 (run/e p INIT-ENV))
```

TEMPLATE

```
;; An AST is one of:
;; …
;; - (mk-lm-ast List<Symbol> AST)
;; …
(struct lm-ast [params body])
```

"Running" Functions?

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


  (defin
    (match p
      …
     [(lm-ast params body) ?? params ?? (run/e body env) ??]

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

Can we "convert" this into a Racket function?

```
;; An AST is one of:
;; …
;; - (mk-lm-ast List<Symbol> AST)
;; …
(struct lm-ast [params body])
```

What should be the "Result" of running a lm function?

```
;; A Result is one of:
;; - Number
;; - ErrorResult
;; - (Racket) Function ???
```

We can't!! (it's not "transparent") (this is what makes FFIs and "multi language" programs complicated) **So we need some other representation**

“Running” Functions?

```
;; An AST is one of:
;; …
;; - (mk-lm-ast List<Symbol> AST)
;; …
(struct lm-ast [params body])
```

Can we “convert” this into a Racket function?

WAIT! Are lm-result and lm-ast the same?

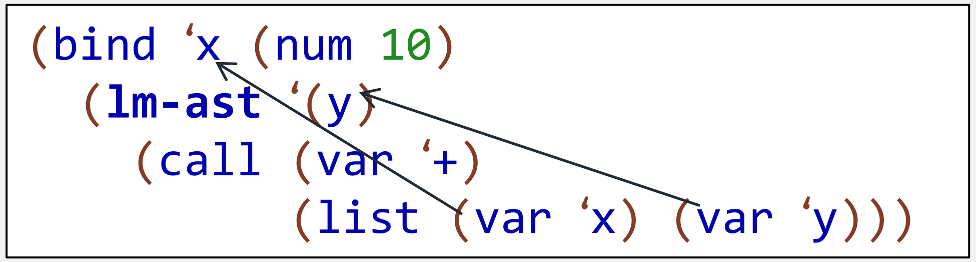
```
;; A Result is one of:
;; - Number
;; - ErrorResult
;; - (Racket) Function
;; - (mk-lm-res List<Symbol> AST ??)
(struct lm-result [params body ??])
```

We can’t!! need some other representation

“Running” Functions? Full example

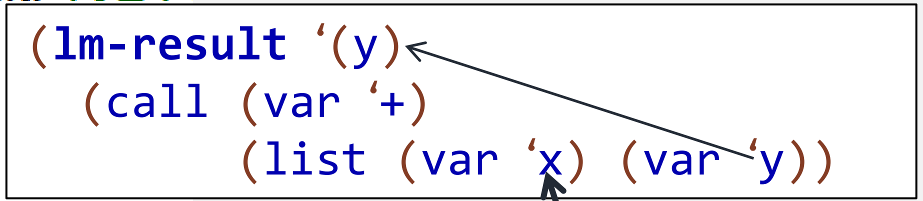
```
(bind [x 10]
  (lm (y) (+ x y)))
```

parse



```
(bind ‘x (num 10)
  (lm-ast ‘(y)
    (call (var ‘+)
          (list (var ‘x) (var ‘y)))
```

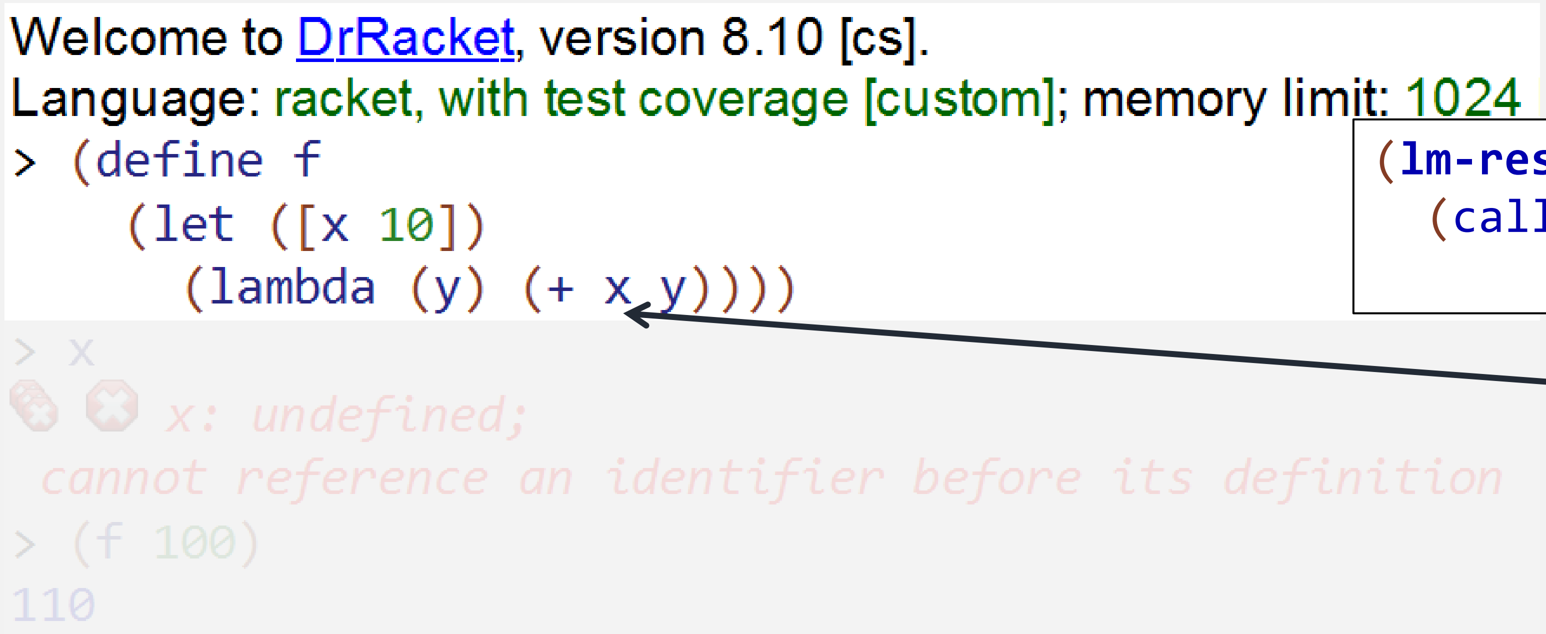
run



```
(lm-result ‘(y)
  (call (var ‘+)
        (list (var ‘x) (var ‘y))
```

Where is the x???

In Racket (with **lambda** and **let**)

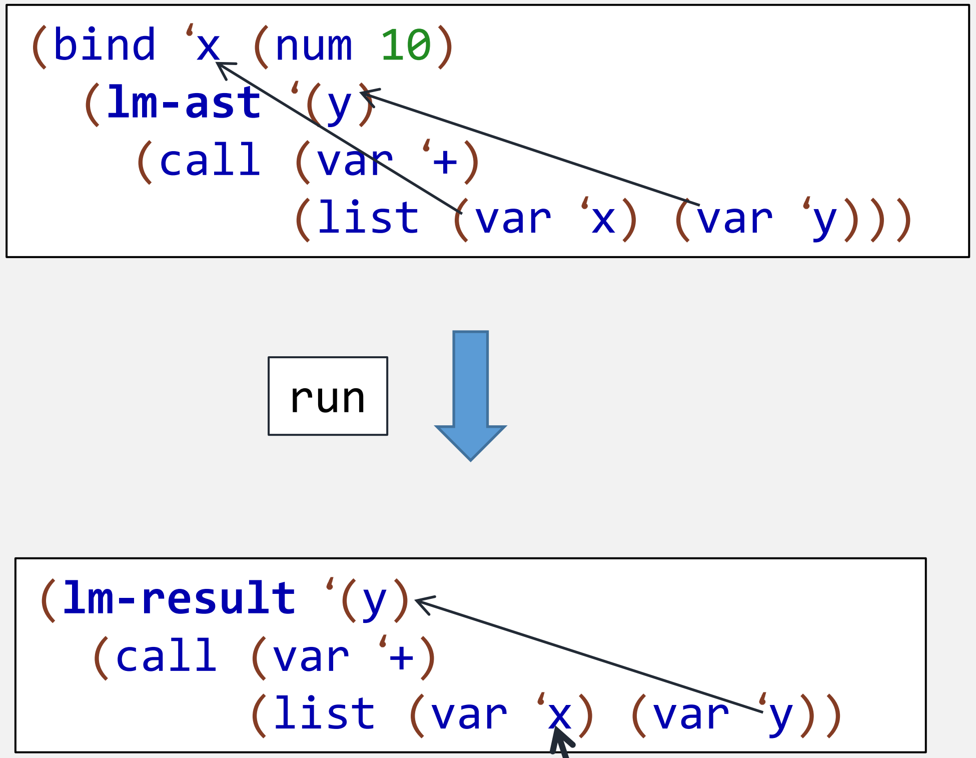


```
Welcome to DrRacket, version 8.10 [cs].
Language: racket, with test coverage [custom]; memory limit: 1024
> (define f
    (let ([x 10])
      (lambda (y) (+ x y))))
> x
x: undefined;
 cannot reference an identifier before its definition
> (f 100)
110
```

“Running” Functions? Full example

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

parse



```
(bind ‘x (num 10)
  (lm-ast ‘(y)
    (call (var ‘+)
          (list (var ‘x) (var ‘y)))
```

run

```
(lm-result ‘(y)
  (call (var ‘+)
        (list (var ‘x) (var ‘y))
```

Where is the x???

`lm-result` **must save** the **x!!**

(how can we **“remember”** the **x**)

“Running” Functions?

```
;; An AST is one of:
;; …
;; - (mk-lm-ast List<Symbol> AST)
;; …
(struct lm-ast [params body])
```

WAIT! Are `lm-result` and `lm-ast` the same?

```
;; A Result is one of:
;; - Number
;; - ErrorResult
;; - (Racket) Function
;; - (mk-lm-res List<Symbol> AST ??)
(struct lm-result [params body ??])
```

“Running” Functions?

Takeaway quiz:
Q: What is the difference between **`lm-ast`** and **`lm-result`**?

A: **`lm-ast`** is AST data
- represents code that a programmer writes;

`lm-result` is `Result` data
- represents result of running the program

(importantly contains **environment** for variables that are not fn parameters)

An `lm` Function `Result` needs an extra environment
(for the non-argument variables used in the body!)

```
;; - ErrorResult
;; - (Racket) Function
;; - (mk-lm-res List<Symbol> AST Env)
(struct lm-result [params body env])
```

“Running” Functions?

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


  (defi
    (match p
        …
     [(lm-ast params body) ?? params ?? (run/e body env) ??]

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

Can we “convert” this into a Racket function?

```
;; An AST is one of:
;; …
;; - (mk-lm-ast List<Symbol> AST)
;; …
(struct lm-ast [params body])
```

What should be the “Result” of running a function?

```
;; A Result is one of:
;; - Number
;; - ErrorResult
;; - (Racket) Function ???
```

We can’t!! need some other representation

“Running” Functions?

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


  (define (run/e p env)
    (match p
        …
     [(lm-ast params body) ?? params ?? (run/e body env) ??]

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

```
;; An AST is one of:
;; …
;; - (mk-lm-ast List<Symbol> AST)
;; …
(struct lm-ast [params body])
```

What should be the “Result” of running a function?

```
;; A Result is one of:
;; - Number
;; - ErrorResult
;; - (Racket) Function
;; - (mk-lm-res List<Symbol> AST Env)
(struct lm-result [params body env])
```

Result of “Running” a Function

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



  (define (run/e p env)
    (match p
        ...
     [(lm-ast params body) (mk-lm-res params body env)]
        ...
     ))
 (run/e p INIT-ENV))
```

body won’t get “run” until the function is called

Save the current **env**

“code”!

```
;; A Result is one of:
;; …
;; - (mk-lm-res List<Symbol> AST Env)
(struct lm-result [params body env])
```

Previously

"Running" Function Calls: Revisited

How do we actually run the function?

```
;; A Result is one of:
;; - Number
;; - ErrorResult
;; - (Racket) Function
```

???

Runs a Racket function

```
(define (run p)


  (define (run/e p env)
    (match p
         ...
      [(call fn args) (apply
                        (run/e fn env)
                        (map (curryr run/e env) args))]

         ...
      ))
 (run/e p INIT-ENV))
```

Does this work???

"Running" Function Calls: Revisited

How do we actually run the function?

```
;; A Result is one of:
;; - Number
;; - ErrorResult
;; - (Racket) Function
;; - (mk-lm-res List<Symbol> AST Env)
(struct lm-result [params body env])
```

```
(define (run p)


  (define (run/e p env)
    (match p
       ...
     [(call fn args) (450apply
                      (run/e fn env)
                      (map (curryr run/e env) args))]

       ...
     ))
 (run/e p INIT-ENV))
```

apply doesn't work for lm-result!!
must manually implement "function call"

(this doesn't "work" anymore!)

CS450 Lang “Apply”

```
;; A Result is one of:
;; - …
;; - (Racket) Function
;; - (mk-lm-res List<Symbol> AST Env)
(struct lm-result [params body env])
```

```
;; 450apply : Result List<Result> -> Result
(define (450apply fn args)
  …
)
```

CS450 Lang "Apply"

TEMPLATE

```
;; A Result is one of:
;; - …
;; - (Racket) Function
;; - (mk-lm-res List<Symbol> AST Env)
(struct lm-result [params body env])
```

```
;; 450apply : Result List<Result> -> Result
(define (450apply fn args)
 (match fn
     …
  [(? procedure?)           …           ] ;; racket function
  [(lm-result params body env)       ;; user-defined function
         …    params           …          body        …            env]))
```

CS450 Lang "Apply"

TEMPLATE: mutually referential data and template calls!

```
;; A Result is one of:
;; - …
;; - (Racket) Function
;; - (mk-lm-res List<Symbol> AST Env)
(struct lm-result [params body env])
```

```
;; 450apply : Result List<Result> -> Result
(define (450apply fn args)
 (match fn
     …
  [(? procedure?)       …        ] ;; racket function
  [(lm-result params body env)      ;; user-defined function
       …     params    …      (ast-fn body … ) … (env-fn env … ) … ]))
```

```
env-add : Env Var Result -> Env
```

CS450 Lang “Apply”

```
;; A Result is one of:
;; - …
;; - (Racket) Function
;; - (mk-lm-res List<Symbol> AST Env)
(struct lm-result [params body env])
```

```
;; 450apply : Result List<Result> -> Result
(define (450apply fn args)
 (match fn
     …
  [(? procedure?)        …        ] ;; racket function
  [(lm-result params body env)      ;; user-defined function
     …    (ast-fn body … ) … (env-add env ?? args params ?? ) … ]))
```

Wait, these are lists

```
env-add : Env Var Result -> Env
```

CS450 Lang "Apply"

```
;; A Result is one of:
;; - …
;; - (Racket) Function
;; - (mk-lm-res List<Symbol> AST Env)
(struct lm-result [params body env])
```

(so this function should be inside run)

```
;; 450apply : Result List<Result> -> Result
(define (450apply fn args)
 (match fn
     …
  [(? procedure?)          …          ] ;; racket function
  [(lm-result params body env)        ;; user-defined function
       …     (ast-fn body … ) … (foldl env-add env params args) … ]))
```

Saved "code"!

these are lists

`run/e : AST Env -> Result`

CS450 Lang “Apply”

```
;; A Result is one of:
;; - …
;; - (Racket) Function
;; - (mk-lm-res List<Symbol> AST Env)
(struct lm-result [params body env])
```

```
;; 450apply : Result List<Result> -> Result
(define (450apply fn args)
 (match fn
                                 ???
     …
  [(? procedure?)              …             ] ;; racket function
  [(lm-result params body env)      ;; user-defined function
   (run/e body (foldl env-add env params args))]))
```

run/e : AST Env -> Result

CS450 Lang “Apply”

```
;; A Result is one of:
;; - …
;; - (Racket) Function
;; - (mk-lm-res List<Symbol> AST Env)
(struct lm-result [params body env])
```

```
;; 450apply : Result List<Result> -> Result
(define (450apply fn args)
 (match fn
     …
  [(? procedure?) (apply fn args)] ;; racket function
  [(lm-result params body env)     ;; user-defined function
   (run/e body (foldl env-add env params args))]))
```

Runs a Racket function

WAIT! What if the the number of params and args don’t match!

CS450 Lang "Apply"

```
;; 450apply : Result List<Result> -> Result
(define (450apply fn args)
 (match fn
     …
   [(? procedure?) (apply fn args)] ;; racket function
   [(lm-result params body env)      ;; user-defined function
    (if (= (length params) (length args))
        (run/e body (foldl env-add env params args))
        …    ]))
```

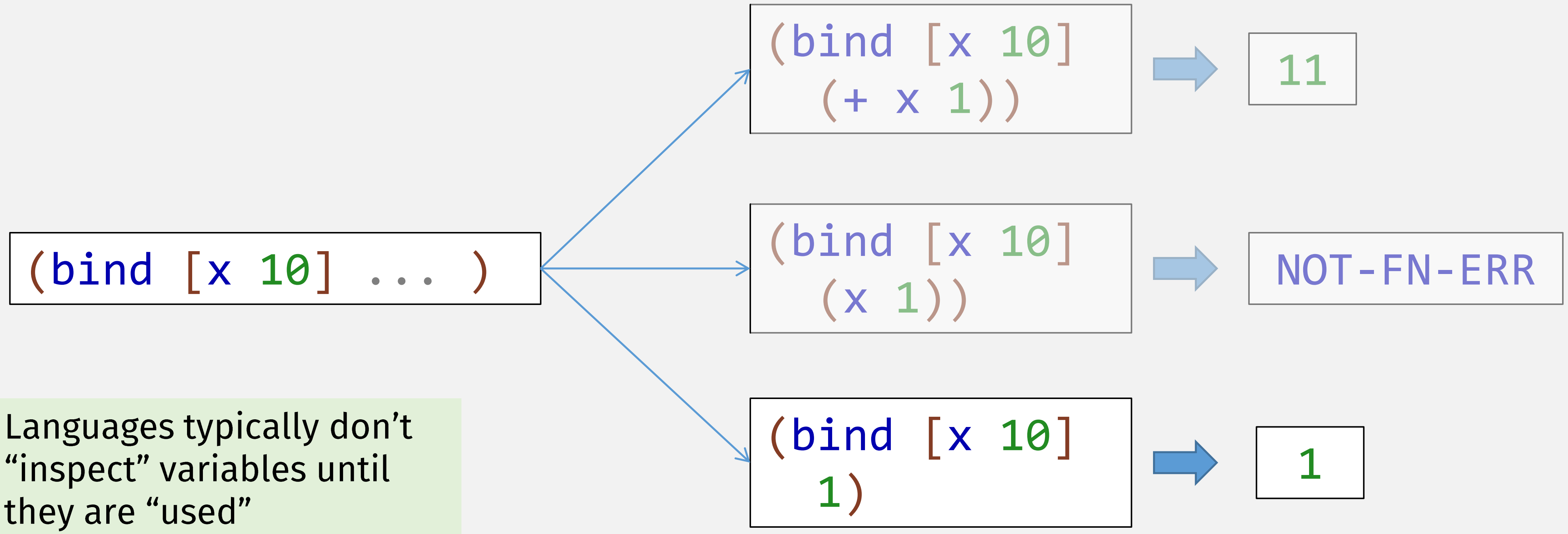
CS450 Lang “Apply”: arity error

```
;; 450apply : Result List<Result> -> Result
(define (450apply fn args)
 (match fn
     …
  [(? procedure?) (apply fn args)] ;; racket function
  [(lm-result params body env)     ;; user-defined function
   (if (= (length params) (length args))
       (run/e body (foldl env-add env params args))
       ARITY-ERROR)]))
```

```
;; An ErrorResult is one of:
;; - UNDEFINED-ERROR
;; - NOT-FN-ERROR
;; - ARITY-ERROR
```

```
;; A Result is one of:
;; - Number
;; - ErrorResult
;; - FnResult
```

Interlude: Error Propagation Example



```
(bind [x 10] ... )
```

```
(bind [x 10]
  (+ x 1))
```

```
11
```

```
(bind [x 10]
  (x 1))
```

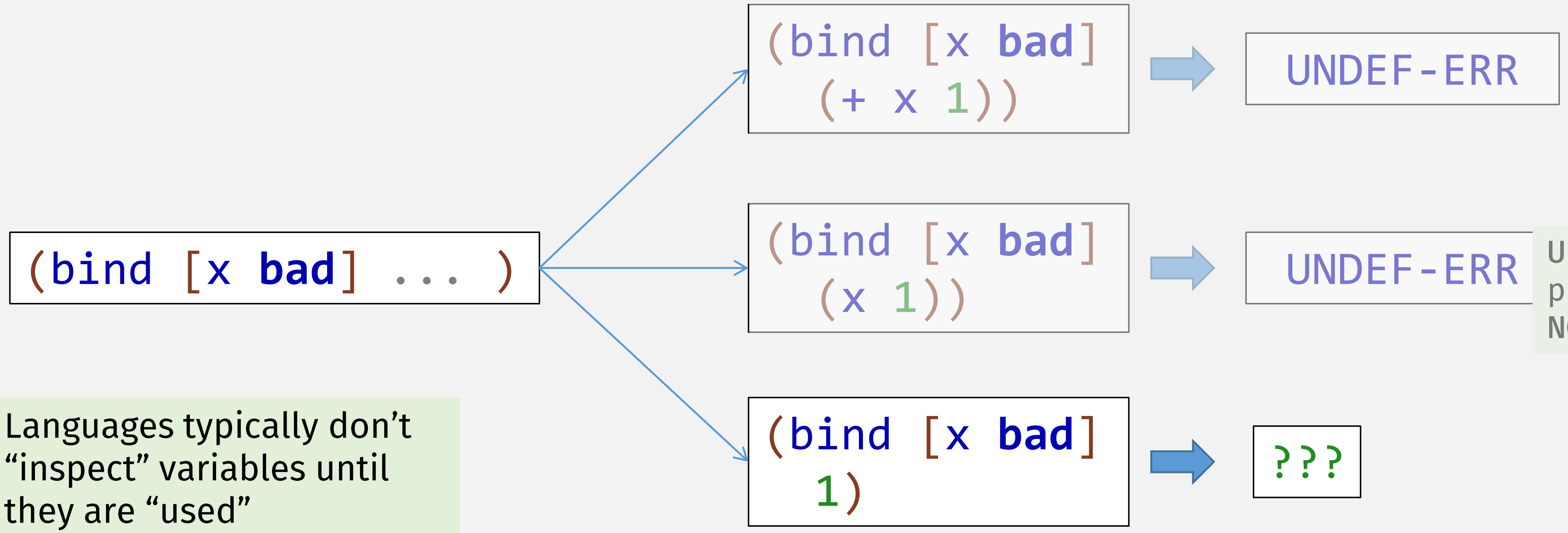
```
NOT-FN-ERR
```

```
(bind [x 10]
  1)
```

```
1
```

Languages typically don't "inspect" variables until they are "used"

Interlude: Error Propagation Example



```
(bind [x bad] ... )
```

```
(bind [x bad]
  (+ x 1))
```

```
UNDEF-ERR
```

```
(bind [x bad]
  (x 1))
```

```
UNDEF-ERR
```

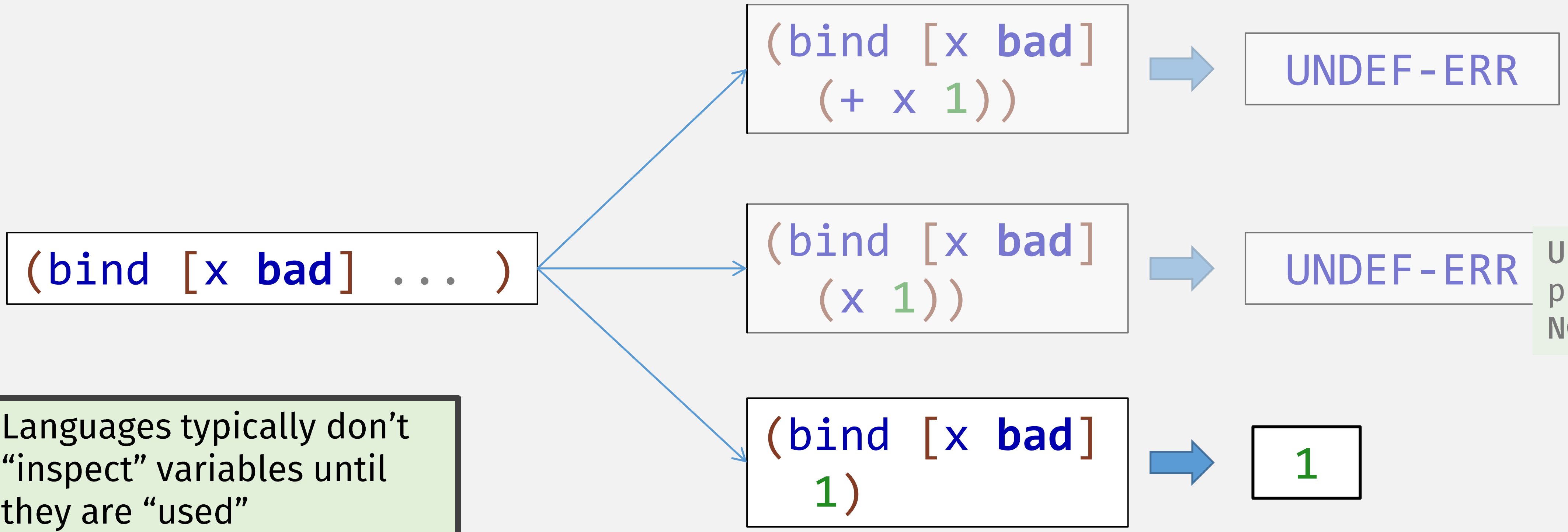
`UNDEF-ERR` has precedence over `NOT-FN-ERR`

```
(bind [x bad]
  1)
```

```
???
```

Languages typically don't "inspect" variables until they are "used"

Interlude: Error Propagation Example



```
(bind [x bad] ... )
```

```
(bind [x bad]
  (+ x 1))
```
→
```
UNDEF-ERR
```

```
(bind [x bad]
  (x 1))
```
→
```
UNDEF-ERR
```

UNDEF-ERR has precedence over NOT-FN-ERR

```
(bind [x bad]
  1)
```
→
```
1
```

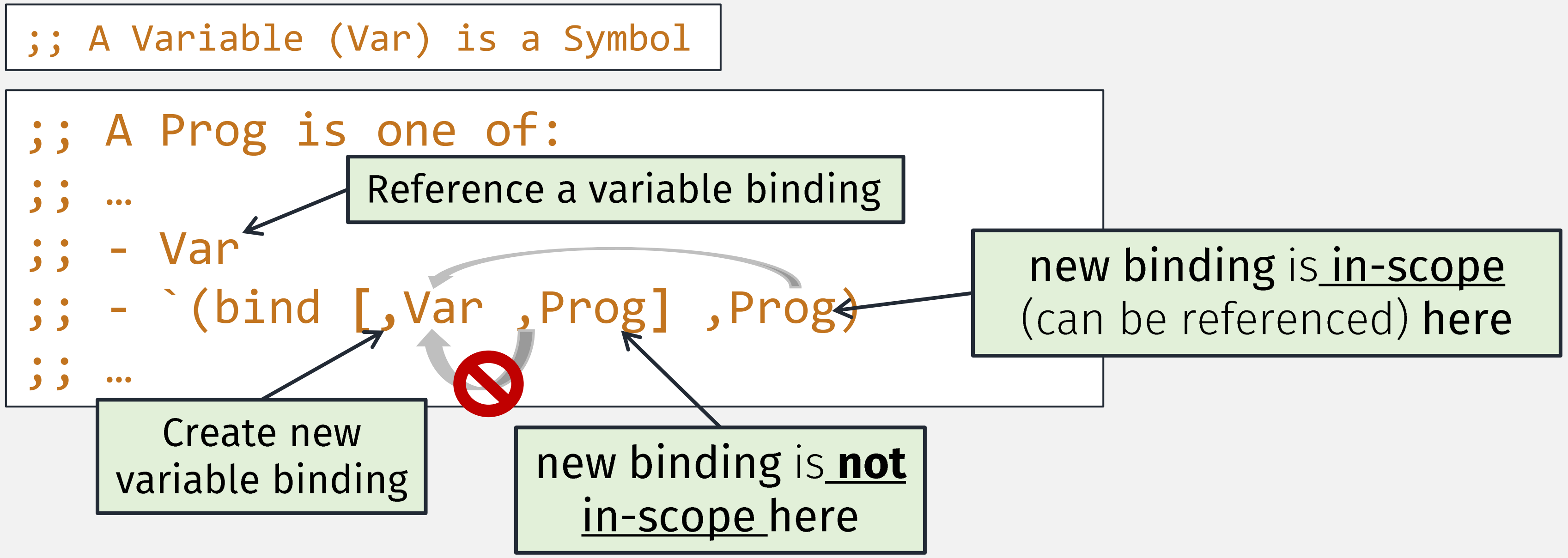
Languages typically don't "inspect" variables until they are "used"

We will choose to ignore err

To avoid complexity of inspecting variables

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 <u>in-scope</u> (can be referenced) **here**

Create new variable binding

new binding is <u>**not** in-scope</u> 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 call

function

arguments

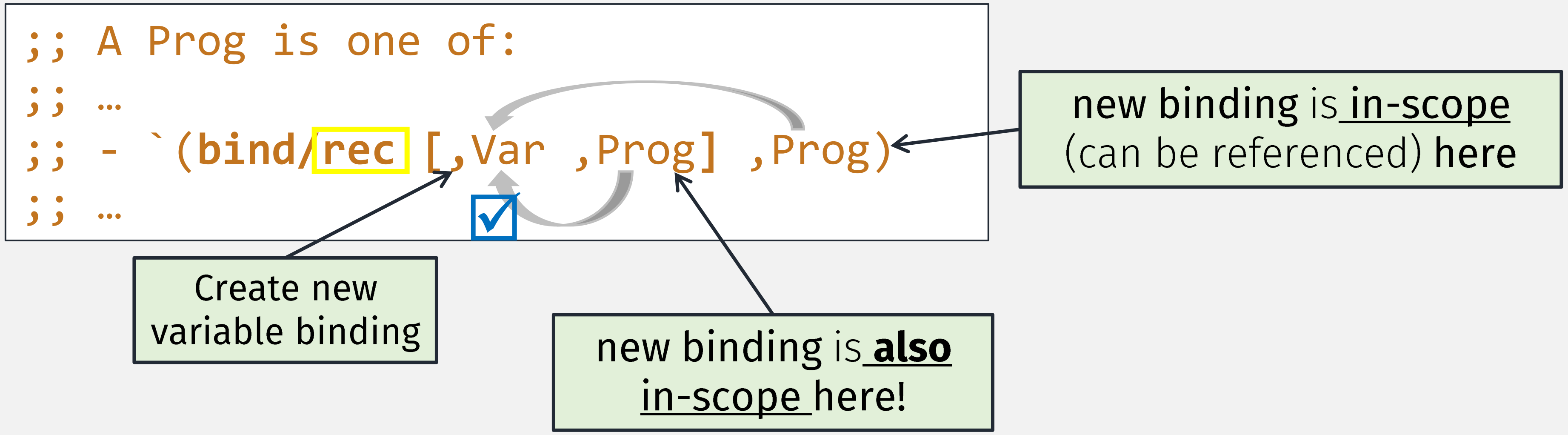
```
(check-equal?
  (eval450
   ‘(bind [f (lm (x) (+ x 4))]
      (f 6)))
   10 )
```

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)
;; …
```

Create new variable binding

new binding is <u>**also** in-scope</u> here!

new binding is <u>in-scope</u> (can be referenced) **here**

Racket recursive function examples

Recursive call

RACKET

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

Equivalent to ...

RACKET

```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)

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 – recursive var practice

Use Racket `letrec` + `lambda` (but not `define`) to write the following recursive functions

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

- Look it up in prev lecture if you don't know any of these
- Write 2 tests to make sure they "work"
  - (tests need to be inside body of `letrec`)