UMass Boston Computer Science

CS450 High Level Languages

Implementing Lambda Functions

Thursday, April 16, 2026

Logistics

- HW 9 in
  - ~~due: Tues 4/14 11am EST~~
  - ~~due: Thur 4/16 11am EST~~

- HW 10 out
  - ~~due: Tues 4/21 11am EST~~
  - due: Thur 4/23 11am EST

Last Time

Function Application in CS450 Lang

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

function

arguments

"rest" arg (like cons)

Specifies arbitrary number of args

Function Application in CS450 Lang: Examples

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

function

arguments

```
(fnapp + 1 2)
```

Programmers should <u>not</u> need to **write** the **explicit "fnapp"!**

Function Application in CS450 Lang: Examples

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

```
(fnapp + 1 2)
```

Function Application in CS450 Lang: Examples

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

"rest" arg (like cons)

```
(+ 1 2)
```

Function Application in CS450 Lang: Examples

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

Can now **go back to** (simpler?) **cons notation**

```
(+ 1 2)
```

Must be careful when parsing this!

Function call case (must be last, why?)

```
(bind 1 2)
```

should <u>not</u> be parsed as function call!

```
(define/contract (parse p)
  (-> Program? AST?)
  (match p
        ...
   [`(bind [,(and (? symbol?) x) ,e] ,bod) ... ]
   [`(bind . ,_)
     (raise-syntax-error 'parse "invalid bind syntax"
        #:exn exn:fail:syntax:cs450)    ]

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

Raise **bind** parse error here

(bind 1 2)

should not be parsed as function call!

(another case below should not match this error case successfully)

Function Application in CS450 Lang

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

parse

```
;; An AST is one of:
;; …
;; - (mk-appl AST List<AST>)
;; …
(struct appl [fn args])
```

“Running” Function Calls

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


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

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

```
;; An AST is one of:
;; …
;; - (mk-appl AST List<AST>)
;; …
(struct appl [fn args])
```

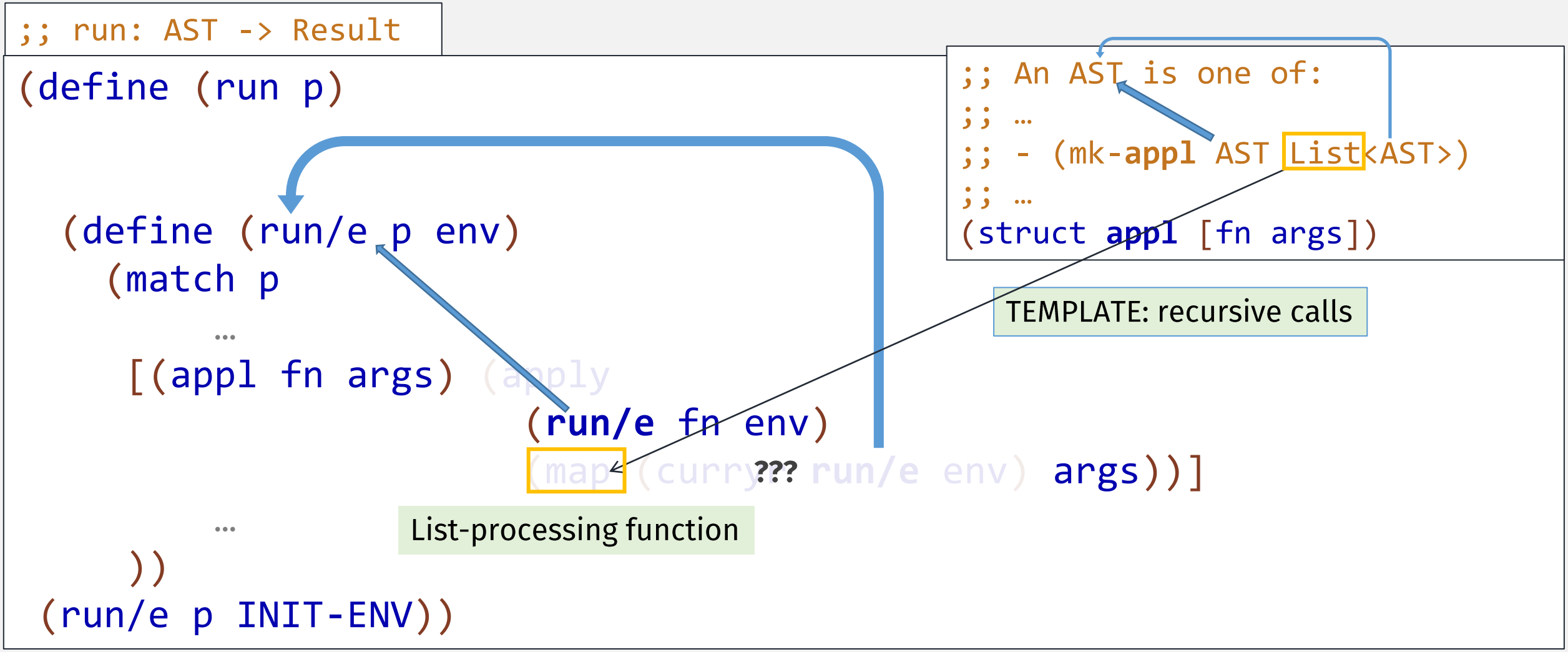
“Running” Function Calls

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


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

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

```
;; An AST is one of:
;; …
;; - (mk-appl AST List<AST>)
;; …
(struct appl [fn args])
```

TEMPLATE: extract pieces of compound data

“Running” Function Calls



“Running” Function Calls

How do we actually run the function?

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

```
(define (run p)


  (define (run/e p env)
    (match p
          …
      [(appl fn args) ???
                      (run/e fn env)
                      (map (curryr run/e env) args)]
          …
      ))
  (run/e p INIT-ENV))
```

Runs a Racket function

function

List of args

Does this work?

“Running” Non-Functions

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

```
(define (run p)


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

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

Example: `(eval450 '(10 10)) ; apply non-fn`

“Running” Non-Functions

```
;; A Result is one of:
;; - Number
;; - UNDEFINED-ERROR
;; - NON-FUNCTION-ERROR
;; - (Racket) Function
```

```
(define (run p)


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

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

Example: `(eval450 '(10 10)) ; apply non-fn`

Function application for CS450 Lang

```
;; A Result is one of:
;; - Number
;; - UNDEFINED-ERROR
;; - NON-FUNCTION-ERROR
;; - (Racket) Function
```

```
;; 450apply : Result Listof<Result> -> Result
```

```
(define (450apply fn args)
  (cond
   [(number? fn)                ... ]
   [(UNDEFINED-ERROR? fn)       ... ]
   [(NON-FUNCTION-ERROR? fn)    ... ]
   [(procedure? fn)             ... ]))
```

TEMPLATE?

Function application for CS450 Lang

```
;; A Result is one of:
;; - Number
;; - UNDEFINED-ERROR
;; - NON-FUNCTION-ERROR
;; - (Racket) Function
```

```
;; 450apply : Result Listof<Result> -> Result
```

```
(define (450apply fn args)
  (cond
   [(number? fn) NON-FUNCTION-ERROR]
   [(UNDEFINED-ERROR? fn)     ... ]
   [(NON-FUNCTION-ERROR? fn) ... ]
   [(procedure? fn)           ... ]))
```

Function application for CS450 Lang

```
;; A Result is one of:
;; - Number
;; - UNDEFINED-ERROR
;; - NON-FUNCTION-ERROR
;; - (Racket) Function
```

```
;; 450apply : Result Listof<Result> -> Result
```

```
(define (450apply fn args)
  (cond
   [(number? fn) NON-FUNCTION-ERROR]
   [(UNDEFINED-ERROR? fn)     ... ]
   [(NON-FUNCTION-ERROR? fn) ... ]
   [(procedure? fn) (apply fn args)]))
```

Now this works

Function application for CS450 Lang

```
;; A Result is one of:
;; - Number
;; - UNDEFINED-ERROR
;; - NON-FUNCTION-ERROR
;; - (Racket) Function
```

```
;; 450apply : Result Listof<Result> -> Result
```

```
(define (450apply fn args)
  (cond
   [(number? fn) NON-FUNCTION-ERROR]
   [(UNDEFINED-ERROR? fn)     ... ]
   [(NON-FUNCTION-ERROR? fn) NON-FUNCTION-ERROR]
   [(procedure? fn) (apply fn args)]))
```

Function application for CS450 Lang

```
;; A Result is one of:
;; - Number
;; - UNDEFINED-ERROR
;; - NON-FUNCTION-ERROR
;; - (Racket) Function
```

```
;; 450apply : Result Listof<Result> -> Result
```

```
(define (450apply fn args)
  (cond
    [(number? fn) NON-FUNCTION-ERROR]
    [(UNDEFINED-ERROR? fn) UNDEFINED-ERROR]
    [(NON-FUNCTION-ERROR? fn) NON-FUNCTION-ERROR]
    [(procedure? fn) (apply fn args)]))
```

UNDEFINED should have precedence over NON-FN-ERR

Combine cases?

Function application for CS450 Lang

```
;; A Result is one of:
;; - Number
;; - UNDEFINED-ERROR
;; - NON-FUNCTION-ERROR
;; - (Racket) Function
```

Add **ARITY-ERROR** ???

```
;; 450apply : Result Listof<Result> -> Result
```

For now, we only use **variable-arity functions**

```
(define (450apply fn args)
  (cond
   [(UNDEFINED-ERROR? fn) UNDEFINED-ERROR]
   [(procedure? fn) (apply fn args)]
   [else NON-FUNCTION-ERROR]))
```

```
(define INIT-ENV
`((+ ,450+)
  (× ,450*)))
```

These should have **"variable arity"** (like Racket +)

Check correct number of arguments???

Combine cases

Don’t need for hw10

Interlude: Variable-arity functions in Racket

Programmer should not be constructing a list

```
;; 450+: List<Result> -> Result ???
```

```
;; 450+: Result … -> Result
(define/contract (450+ . args)
  (-> Result? ... Result? )
    …   )
```

In the function body, **args** is a **list of arguments**

This should now have **“variable arity”** (like **Racket +**)

(compare with JS “variadic” args)

```
function sum(...theArgs) {
  let total = 0;
  for (const arg of theArgs) {
    total += arg;
  }
  return total;
}
```

Not for hw10!

Function Application in CS450 Lang

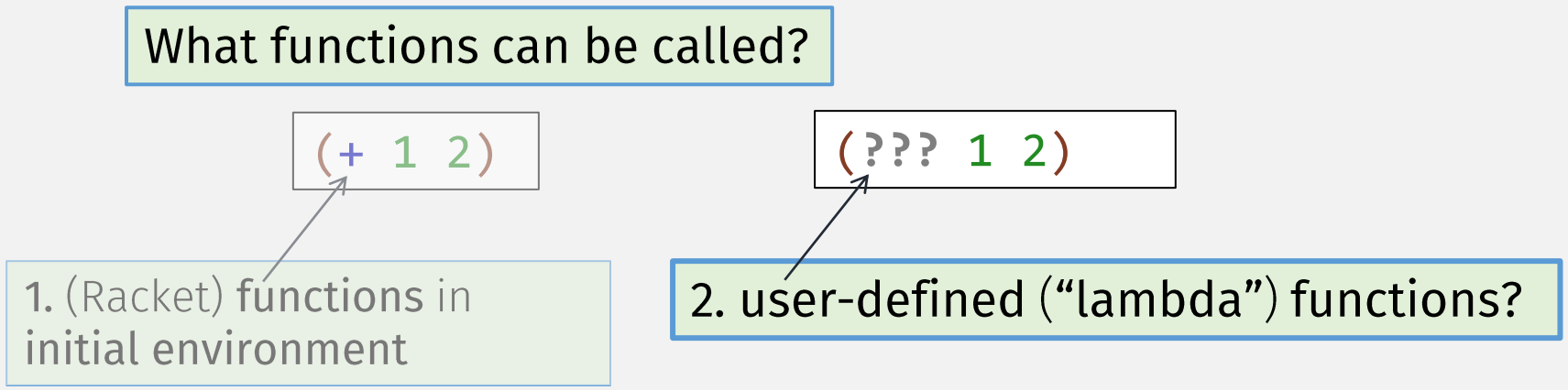
```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - (cons Program List<Program>)
```

Function call case (must be last)

What if a user wants to define their own function!

Next Feature: **Lambdas?**

Function Application in CS450 Lang



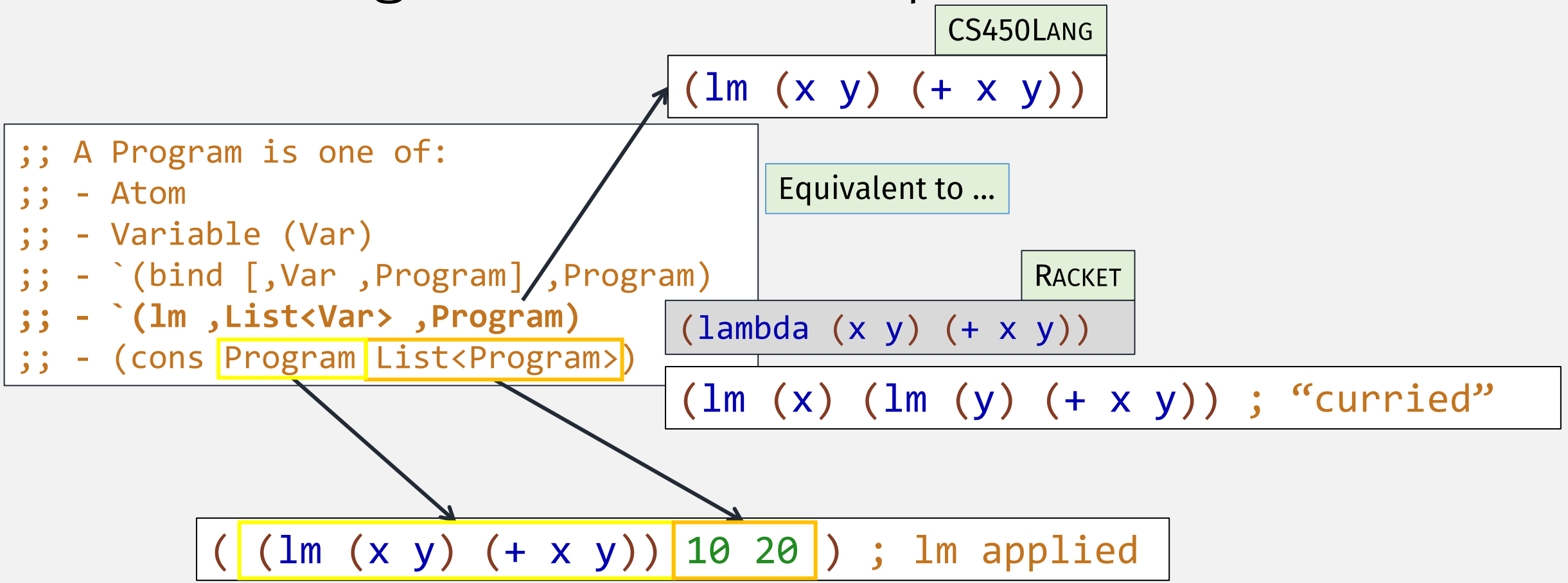
“Lambdas” in CS450 Lang

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

“Lambdas” in CS450 Lang

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

CS450 Lang “Lambda” examples



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

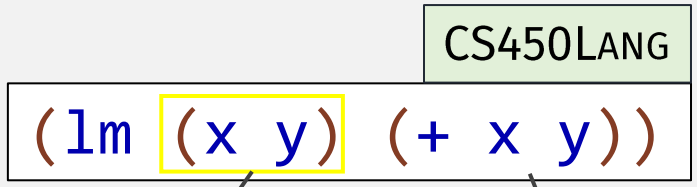
```
;; An AST is one of:
;; …
;; - (mk-lm-ast List<Symbol> AST)
;; …
```

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

```
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
```

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

Parsing “Lambda”



Pattern that matches correct `lm` syntax

```
(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)]))
```

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>)
```

parse

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


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

eval450

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

run

This **represents code** (that has **not been run**)!

CS450 Lang “Lambda” full examples

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

Can reference non-parameter bindings

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

Expression that evaluates to a function result

argument

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

In-class Coding 4/16: `lm` scope examples

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

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

Expression that evaluates to a function result

argument

Come up with some of your own!

```
(check-equal?
  (eval450
   '( (lm (x y) (+ x y))
      10 20 ) )
   30 )
```

Check your understanding in Racket! (using **lambda** and **let**)

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>)
```

parse

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

Why can’t we use a **Racket `lambda`?**

Because this **represents code!**

A **Racket `lambda`** is a **“Result”,** e.g., you can’t **“get”** the **parameters** or the **body code** (it’s not “transparent”)

“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)


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

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

What should be the “Result” of running an lm function?

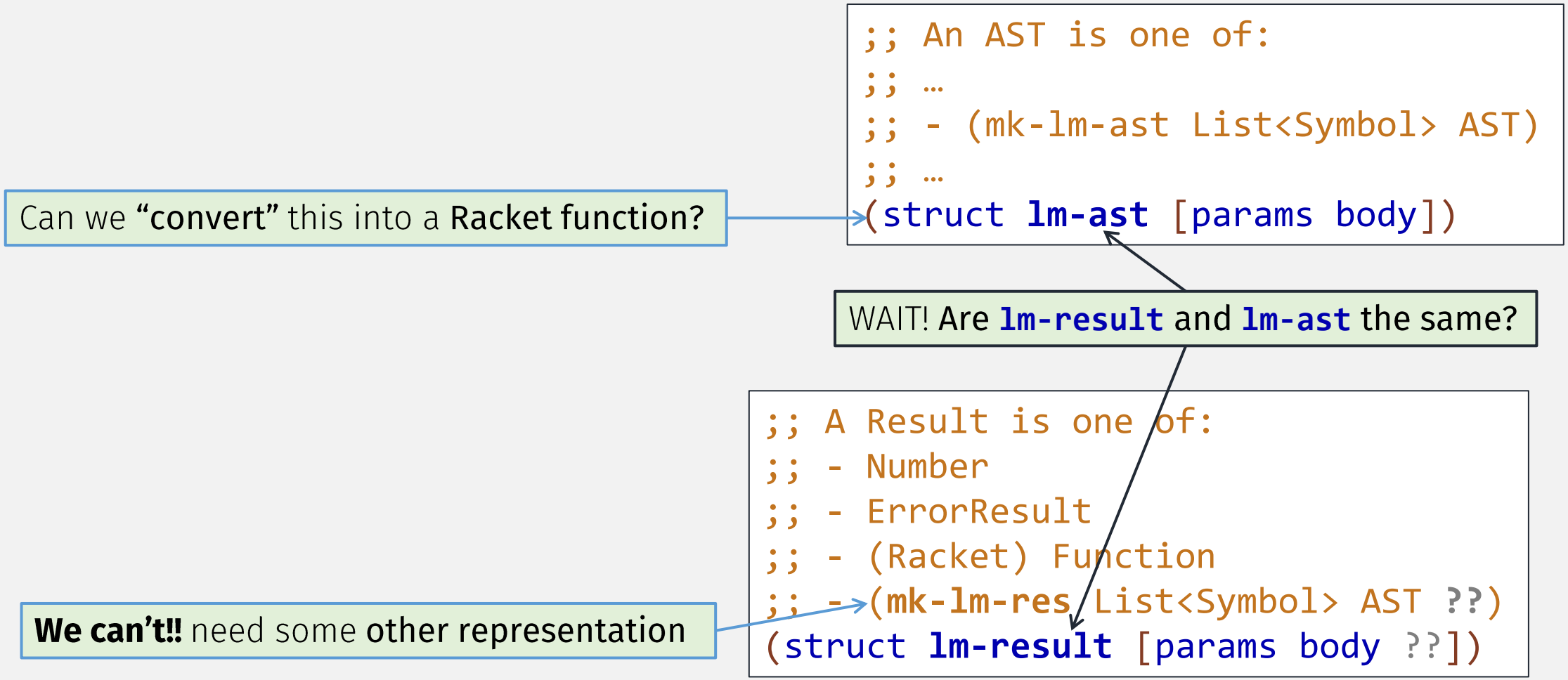
```
;; A Result is one of:
;; - Number
;; - ErrorResult
;; - (Racket) Function ???
```

Can we “convert” a 450lang “lm” AST into a Racket function???

We can’t!! (it’s not “transparent”) (this is what makes FFIs and mixed lang progs complicated) **So we need some other representation**

e.g., can’t call Python function from Java, etc

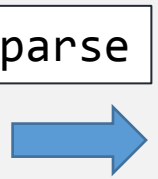
“Running” Functions?



“Running” Functions? Full example

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

parse



???

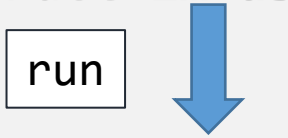
```
(bind 'x (num 10)
  (lm-ast '(y)
    (appl (var '+)
          (list (var 'x) (var 'y)))
```

```
;; - (mk-lm-ast List<Symbol> AST)
;; …
(struct lm-ast [params body])
```

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
```

run



```
sult '(y)
l (var '+)
  (list (var 'x) (var 'y))
```

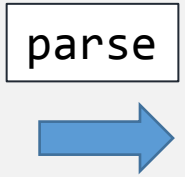
Where is the x???

```
(Racket) Function
(mk-lm-res List<Symbol> AST ??)
t lm-result [params body ??])
```

“Running” Functions? Full example

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

parse



???

```
(bind 'x (num 10)
  (lm-ast '(y)
    (appl (var '+)
          (list (var 'x) (var 'y)))))
```

run

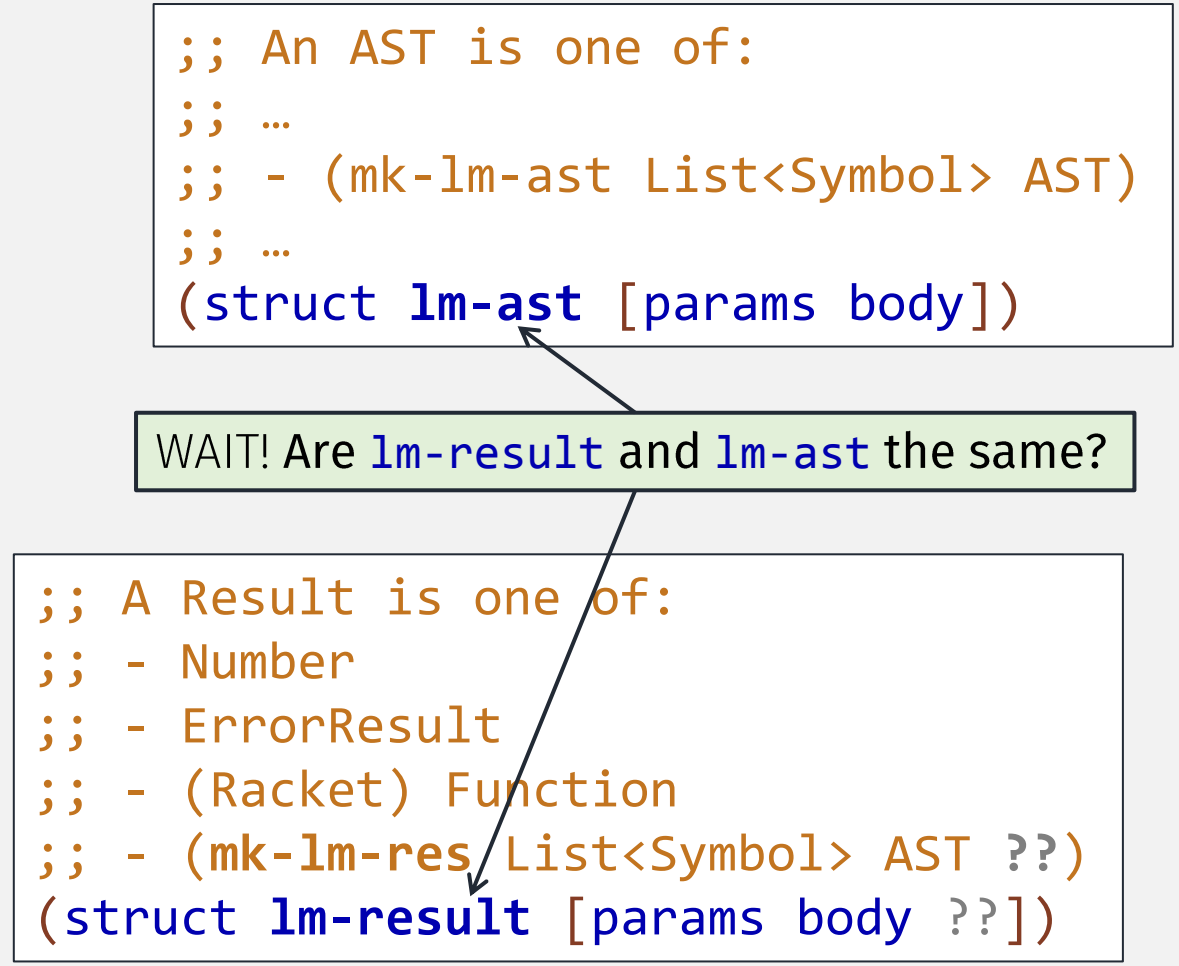
```
(lm-result '(y)
  (appl (var '+)
        (list (var 'x) (var 'y))))
```

Where is the x???

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

(how can we “**remember**” the **x**)

“Running” Functions?



“Running” Functions?

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

```
;; An AST is one of:
;; …
```

A: `lm-ast` is **AST data**
- **represents code** that a **programmer writes** (hasn’t been run!);

`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)


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

```
;; 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 ???
```

Can we “**convert**” a **450lang** “**lm**” **AST** into a **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))
```

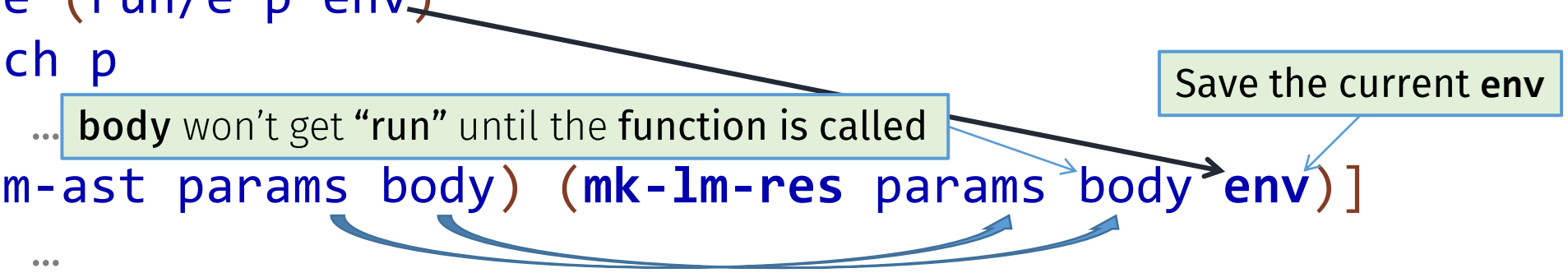
What should be the “**Result**” of running a function?

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

```
;; 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))
```



Previously

Running Function "Calls": Revisited

How do we actually **run** the **function**?

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

```
(define (run p)


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

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

???

Runs a **Racket function**

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
          …
      [(appl 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 **internal** in **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) … ]))
```

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

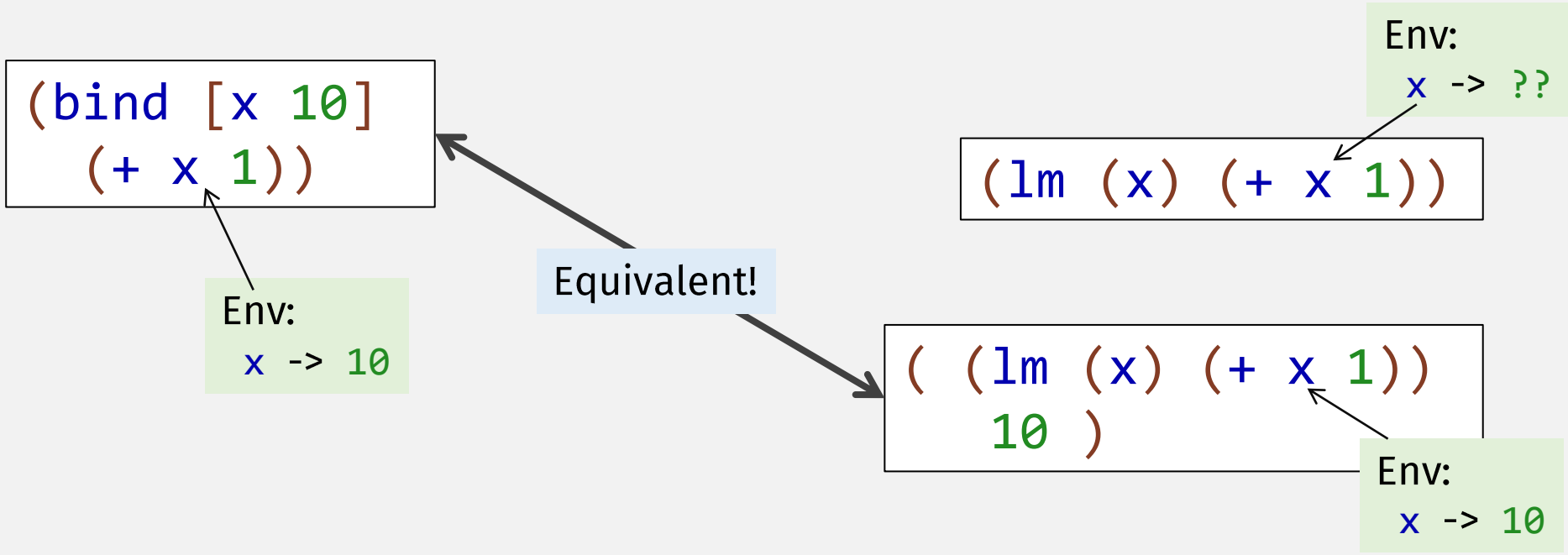
CS450 Lang “Apply”

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

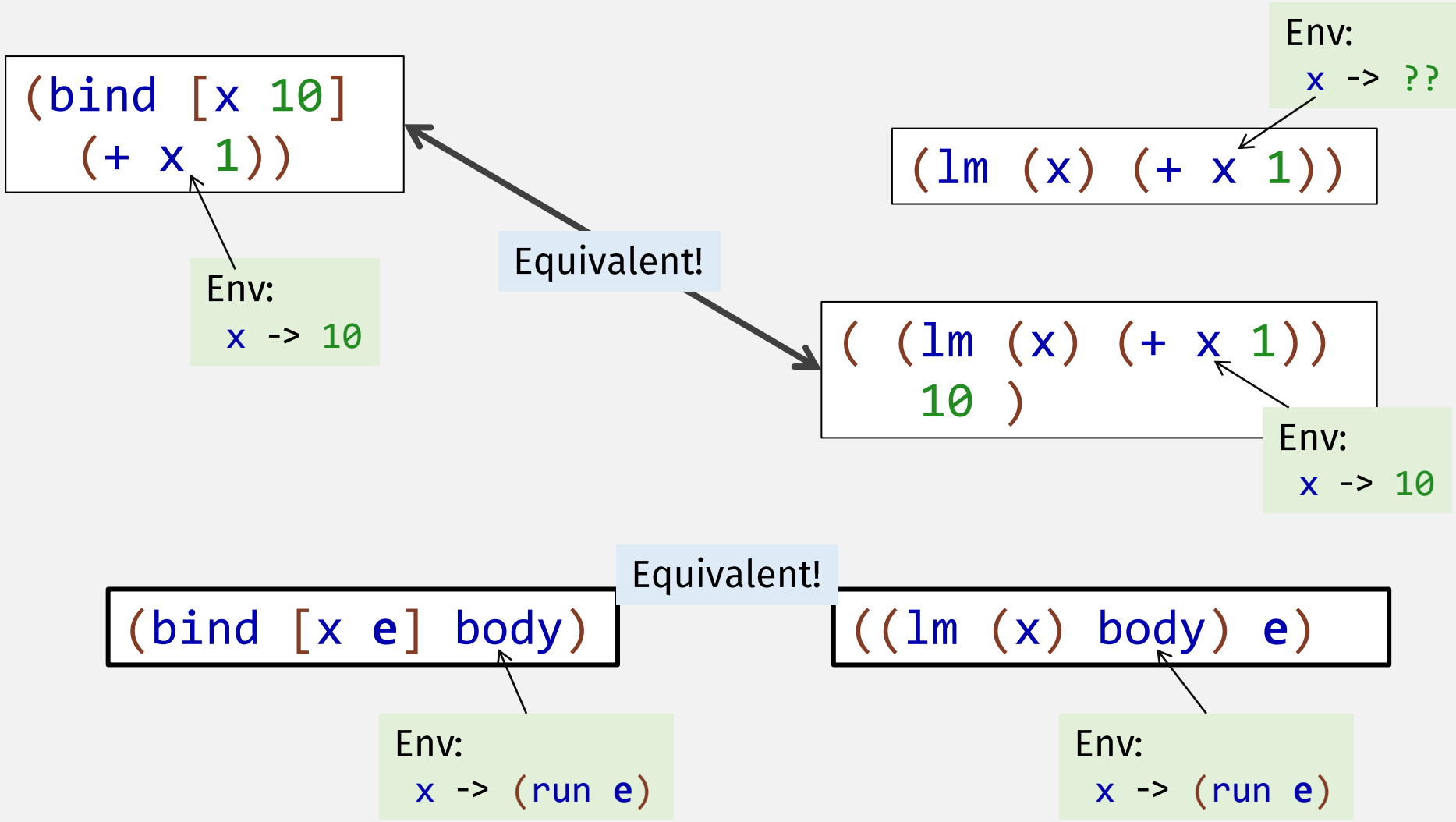
```
;; 450apply : Result List<Result> -> Result
(define (450apply fn args)
 (match fn
      …
  [(? procedure?) (apply fn args)] ;; racket function
  [(lm-result params body env)     ;; user-defined function
   (run/e body (foldl env-add env params args))]))
```

Runs a Racket function

bind, **lm**, and their environments



bind = **lm** + fn call!



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))
       …   ]))
```

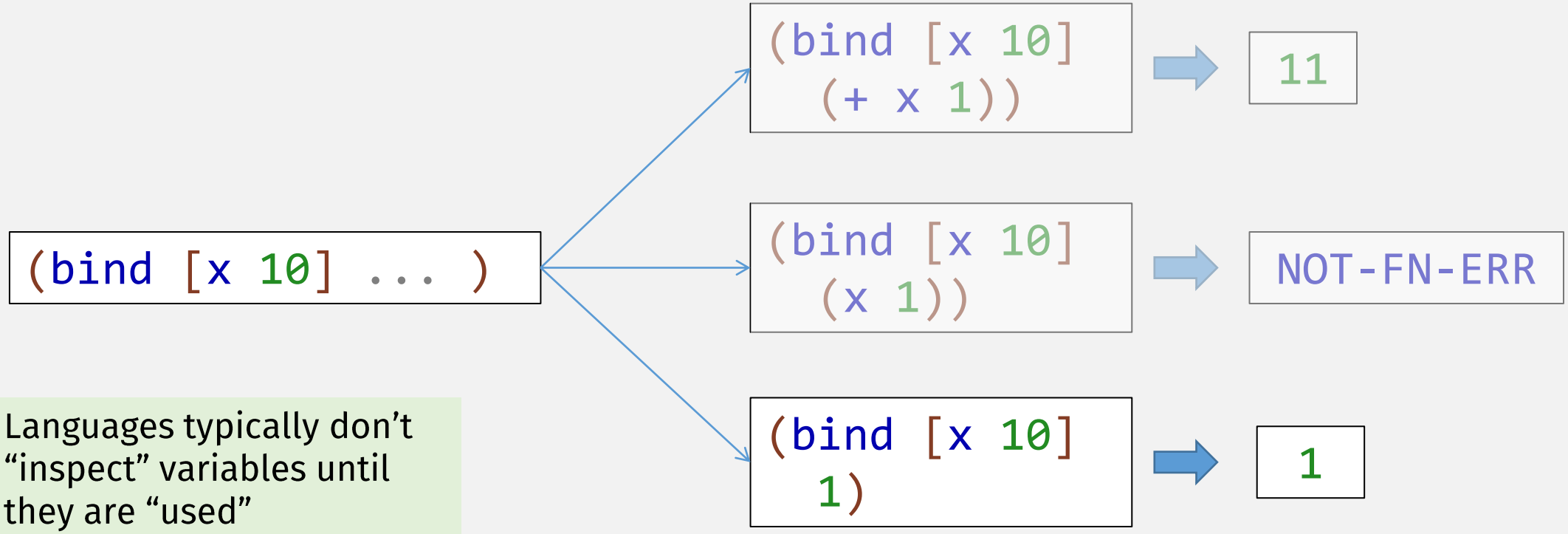
CS450 Lang “Apply”: arity error

```
;; 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
```

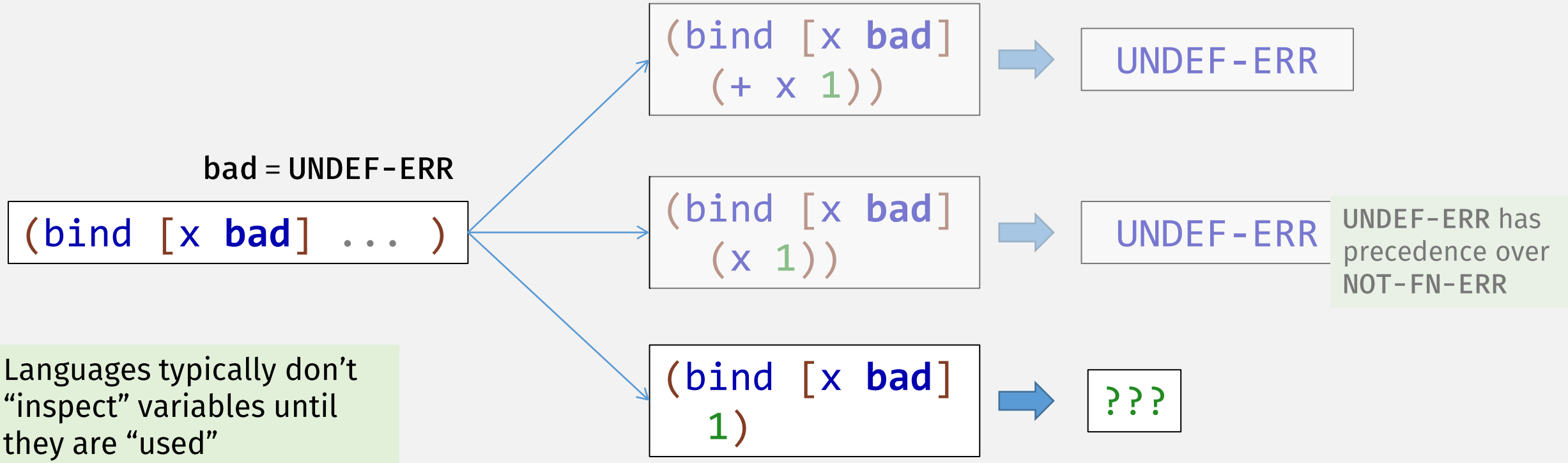
```
;; A Result is one of:
;; - Number
;; - ErrorResult
;; - FnResult
```

Interlude: Error Propagation Example



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

Interlude: Error Propagation Example



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

Not for hw10!

Interlude: Error Propagation Example

bad = UNDEF-ERR

```
(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 before their use

In-class Coding 4/16 `lm` examples

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

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

Expression that evaluates to a function result

argument

```
(check-equal?
  (eval450
   '( (lm (x y) (+ x y))
      10 20 ) )
   30 )
```

Come up with some of your own!
(i.e., not my examples)
(can be error cases, both "syntax" and "result")

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