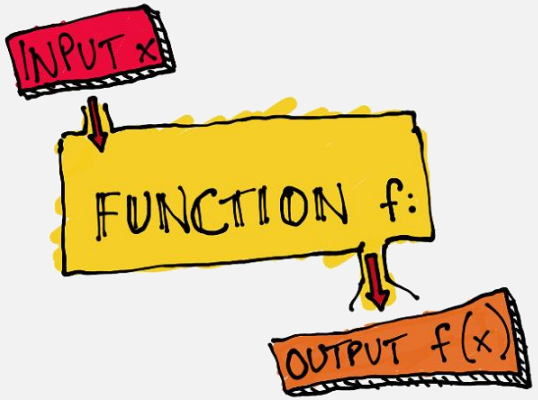
UMass Boston Computer Science

**CS450 High Level Languages**

# Implementing Function Calls

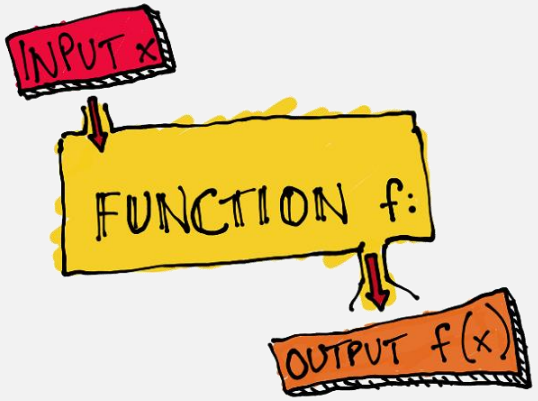
Tuesday, April 14, 2026



# *Logistics*

- HW 9 extended
  - ~~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



# My Code “Works”!

… is not a measure of code quality!

```
(define (mk-Slice/test #:start [start #f]
                       #:stop [stop #f]
                       #:step [step #f])
  (define real-start
    (if (false? start)
        0
        start))
  (define real-stop
    (if (false? start)
        -1
        stop))
  (define real-step
    (if (false? start)
        1
        stop))
  (SliceRange real-start real-stop real-step))
```

1. Doesn’t “work”
2. Is 10x longer than needed

## AI-Generated Code Poses Security, Bloat Challenges

Development teams that fail to create processes around AI-generated code face more technical and security debt as vulnerabilities get replicated.

r/VibeCodersNest • 5mo ago
thoughtfulbear10

### Anyone else’s AI generated codebase slowly turning into chaos?

General Discussion

I’ve been building my app using a mix of Cursor and Claude Artifacts. At first it was clean, but every time I ask for a new feature, the AI rewrites big chunks of the codebase. Sometimes it changes structure, sometimes it adds more dependencies, sometimes it moves things around without warning.

```
(define (mk-Slice/test #:start [start #f]
                       #:stop [stop #f]
                       #:step [step #f])
  (define decoy-start
    (if (false? start)
        100000
        start))
  (define real-stop
    (if (false? start)
        100001
        stop))
  (define real-step
    (if (false? start)
        100002
        stop))
  (define real-start
    (if (= 100000 decoy-start)
        0
        start))
  (define real-stop
    (if (= 100001 decoy-stop)
        -1
        stop))
  (define (= 100002 deocoy-step)
    (if (false? start)
        1
        stop))
  (SliceRange real-start real-stop real-step))
```

This code “works” too!

Last Time

# Adding Variables

Programmer writes:

```
;; A Program is
;; - Atom
;; - Variable
;; - ...
```

```
;; An Variable is a:
;; - Symbol
```

**$Q_1$**: What is the “**meaning**” of a **variable**?

**$A_1$**: Whatever “**value**” it represents

**$Q_2$**: Where **do these “values” come from?**

**$A_2$**: Other **parts of the program!**

```
;; An AST is one of:
;; - …
;; - (mk-var Symbol)
;; …

;; …
(struct vari [name])
```

Hint: Don’t use “**var**” for **struct name** (reserved **Racket match** pattern)

```
;; A Result is one of:
;; - Boolean
;; - String
;; - ???
```

The **run** function needs to “remember” these values ( with an **accumulator**!)

**run**

(JS semantics)

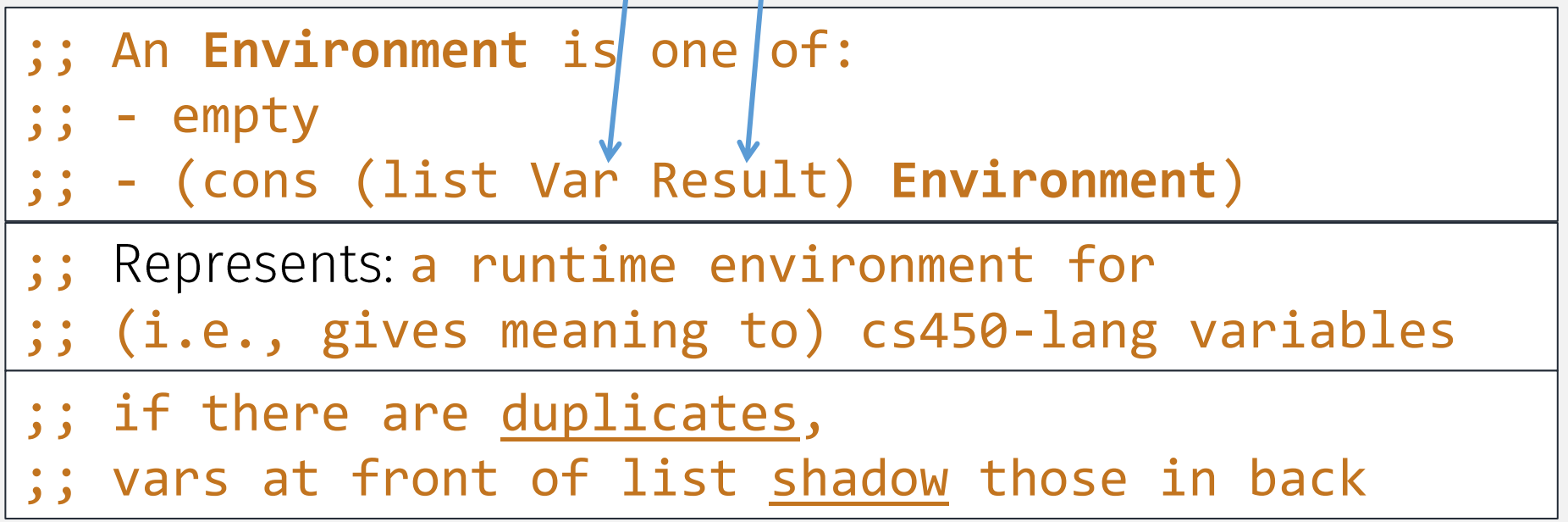
“meaning” of the program

# **run**, with an accumulator

```
;; run: AST -> Result
;; Computes result of running a CS450 Lang program AST
(define (run p)
  ;; accumulator acc : Environment
  ;; invariant: Contains variable values … currently in-scope
  (define (run/acc p acc)
    (match p
      [(num n) n]
      [(add x y) (450+ (run/acc x acc) (run/acc y acc))]))
 (run/acc p  ???  ))
```

# Environments

- A **data structure** that **“associates” two things** (`var`, `val`) **together**
  - E.g., **maps, hashes,** etc
  - We use **list-of-pairs**



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

```
;; Represents: a runtime environment for
;; (i.e., gives meaning to) cs450-lang variables
```

```
;; if there are duplicates,
;; vars at front of list shadow those in back
```

# Environments

- A **data structure** that **“associates” two things** (`var`, `val`) **together**
  - E.g., **maps, hashes,** etc
  - We use **list-of-pairs**

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

- Needed operations:
  - `env-add    : Env Var Result -> Env`
  - `env-lookup : Env Var -> Result`

```
;; A Result is one of:
;; - Number
;; - UNDEFINED-ERROR
```

Outputs **`UNDEFINED-ERROR`** if **`Var`** not in **`Env`**

# **run**, with an Environment accumulator

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
     [(num n) n]
     [(add x y) (450+ (run/env x acc) (run/env y acc))]))
 (run/env p  ???  ))
```

# **run**, with an Environment accumulator

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
      …
      [(vari x) (env-lookup env x)]
      [(bind x e body) … (env-add env x (run/env e env)) …]
      …   ))
 (run/env p  ???  ))
```

Don't name this `var`

# **run**, with an Environment accumulator

**TODO:**
- When are variables “added” to environment???
- What is initial environment?

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
      …
      [(vari x) (env-lookup env x)]
      [(bind ??? body) … (env-add env x ??? e env)) …]
      …  ))
 (run/env p ???  ))
```

# Programs that Add Variables to Environment

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - ?????
```

# Programs that Add Variables to Environment

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

(like "**let**" in other langs)

# Programs that Add Variables to Environment

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

parse

```
;; An AST is one of:
;; - …
;; - (mk-var Symbol)
;; - (mk-bind Symbol AST AST)
;; - …

;; …
(struct vari [name])
(struct bind [var expr body])
;; …
```

Need to be more careful parsing

Valid Program?

```
'(bind)
```

```
'(bind [])
```

```
'(bind [1 2] 3)
```

```
(define/contract (parse p)
  (-> Program? AST?)
  (match p
        ...
   [`(bind [,(and (? symbol?) x) ,e] ,bod) ... ]

        ...


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

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

Bind parse error case

(don’t want another case below to match this error case)

# Programs that Add Variables to Environment

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

parse

```
;; An AST is one of:
;; - …
;; - (mk-var Symbol)
;; - (mk-bind Symbol AST AST)
;; - …

;; …
(struct vari [name])
(struct bind [var expr body])
;; …
```

Need to be more careful parsing

Valid Program?

```
(check-exn exn:fail:syntax:cs450?
    (λ () (eval450 '(bind))))
```

```
(check-exn exn:fail:syntax:cs450?
    (λ () (eval450 '(bind []))))
```

```
(check-exn exn:fail:syntax:cs450?
    (λ () (eval450 '(bind [1 2] 3))))
```

# Programs that Add Variables to Environment

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

parse

```
;; An AST is one of:
;; - …
;; - (mk-var Symbol)
;; - (mk-bind Symbol AST AST)
;; - …

;; …
(struct vari [name])
(struct bind [var expr body])
;; …
```

run

???

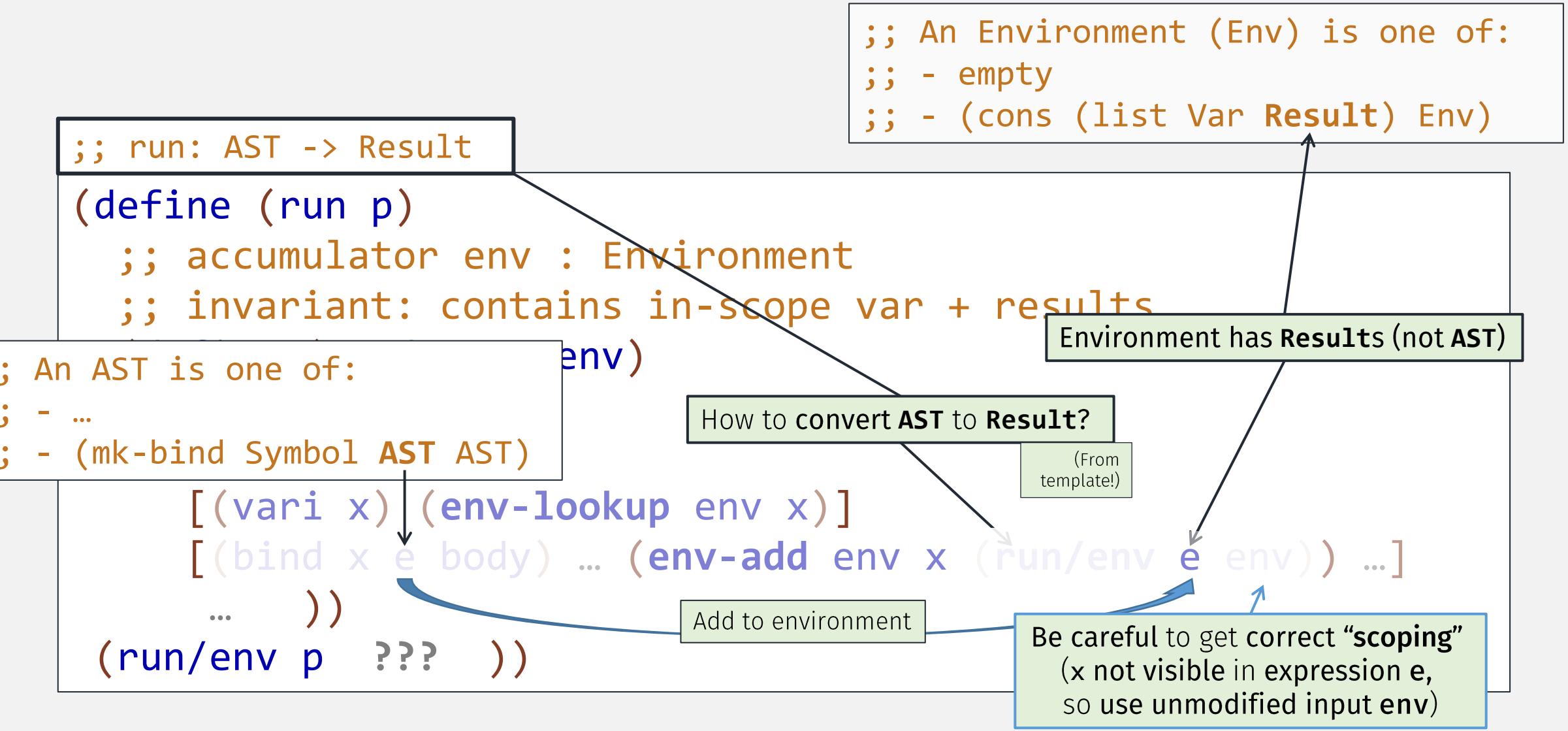
# Bind scoping examples

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

**bind** : want **“lexical”** or **“static”** scoping

Generally accepted to be **“best choice”**
for **programming language design**
(bc it’s determined only by program syntax)

# run, with **bind**



# `run`, with **`bind`**

`run` must produce **`Result`**

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env e env)
    (type-case AST e
      [(vari x) (env-lookup env x)]
      [(bind x e body) ??? (env-add env x (run/env e env)) …]
      …  ))
  (run/env p  ???  ))
```

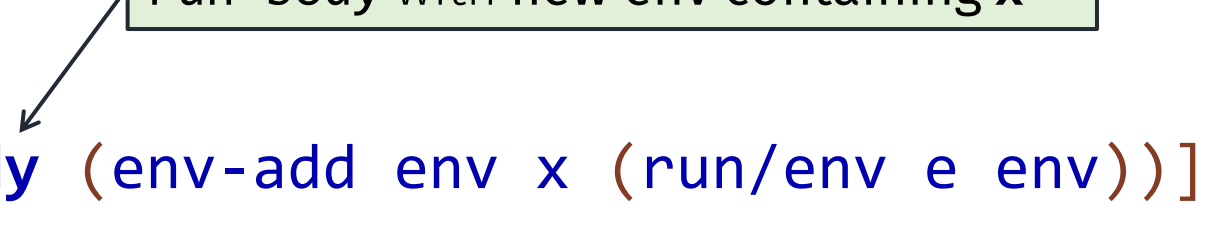
```
; An AST is one of:
; - …
; - (mk-bind Symbol AST AST)
```

# run, with **bind**

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
      …
     [(vari x) (env-lookup env x)]
     [(bind x e body) (run/env body (env-add env x (run/env e env)))]
      …   ))
 (run/env p  ???  ))
```

(From template!)

run body with new env containing **x**



# Initial Environment?

**TODO:**
- ~~When are variables "added" to environment~~
- What is **initial environment?** `empty` (for now)

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
      …
      [(vari x) (env-lookup env x)]
      [(bind x e body) (run/env body (env-add env x (run/env e env)))]
      …   ))
 (run/env p  ???  ))
```

`empty` ??? (for now)

# Adding to Initial Environment?

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
```

These **don't** need to be **separate constructs!**

Put these into **"initial" environment**

# Adding to Initial Environment?

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
```

Put these into "initial" environment

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

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

For Program: +

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

New kind of Result

Maps to our "450+" function

+ variable

# Adding to Initial Environment?

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

But … how do **users** “**call**” these **functions???**

```
(define (run p)

  ;; accumulator env : Environment
  (define (run/e p env)
    (match p
      …
     [(vari x) (lookup env x)]
     [(bind x e body) (run/e body (env-add env x (run/e e env)))]
      … ))
 (run/e p INIT-ENV ))
```

# 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>)
```

```
(+ 1 2)
```

Must be careful when parsing this!

**Function call case** (must be last, why?)

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

```
(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])
```

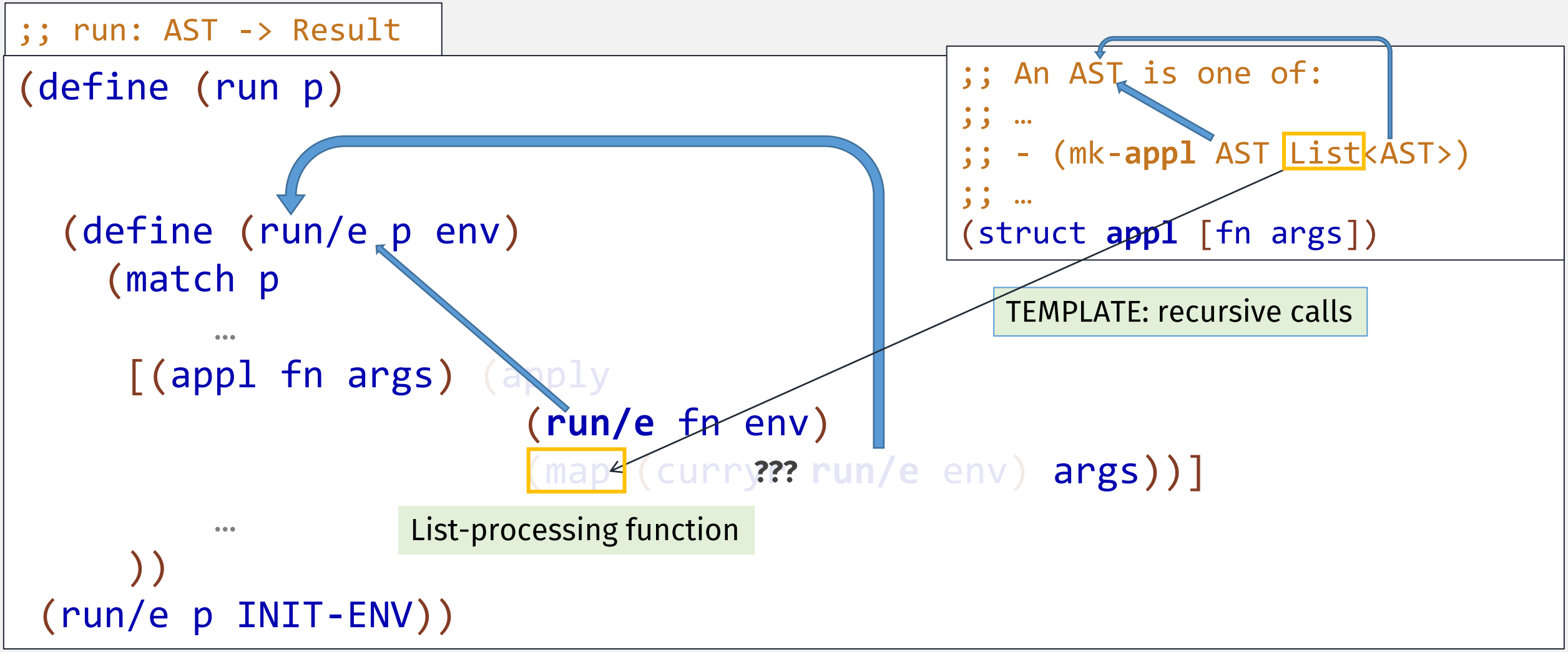
# “Running” Function Calls

```
;; 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))
```

# 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

# *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?**