UMass Boston Computer Science

CS450 High Level Languages

Implementing Lambda

Thursday, April 24, 2025

(don’t need lambda for hw11)

Logistics

- HW 11 out
  - due: Tues 4/29 11am EST

Last Time

“CS450” Lang, with Vars and Fn Calls

Programmer writes:

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

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

Last Time

"CS450" Lang, with Vars and Fn Calls

Programmer writes:

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

parse

```
;; An AST is one of:
;; - …
;; - (mk-var Symbol)
;; - (mk-bind Symbol AST AST)
;; - (mk-call AST List<AST>)
;; - …
;; …
(struct vari [name])
(struct bind [x e body])
(struct call [fn args])
;; …
```

Hint: Don't use "**var**" (reserved for a Racket **match** pattern) for struct name

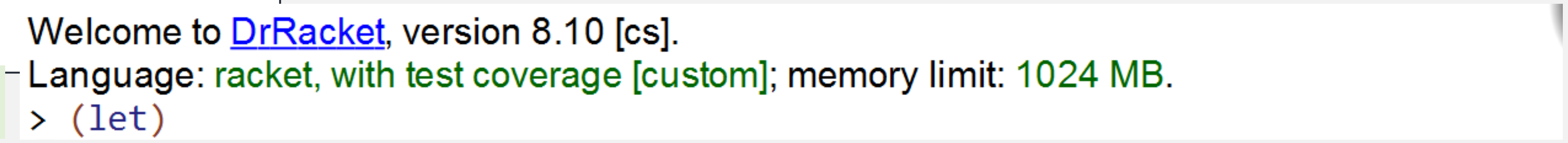
Parsing **bind** programs

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

parse

```
;; An AST is one of:
;; - …
```

Need to be more careful parsing



```
Welcome to DrRacket, version 8.10 [cs].
Language: racket, with test coverage [custom]; memory limit: 1024 MB.
> (let)
```

Parsing **bind** programs

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

Need to be more careful parsing

parse

```
;; An AST is one of:
;; - …
```

Valid Program?

```
‘(bind)
```

```
‘(bind [])
```

```
‘(bind [1 2] 3)
```

Interlude: Racket exceptions

Exceptions are just special structs

Super struct (enables using **exception API**)

```
(struct exn:fail:syntax:cs450 exn:fail:syntax [])
```

```
(define/contract (parse p)
  (-> Program? AST?)
  (match p
   [(? atom?) (parse-atom p)]
     ...
   [`(,fn . ,args) ... ]
   [_ (error … )]))
```

Interlude: Racket exceptions

Exceptions are just special structs

Super struct (enables using **exception API**)

```
(struct exn:fail:syntax:cs450 exn:fail:syntax [])
```

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

Parsing **bind** programs

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


      ...


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

Parsing **bind** programs

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

Bind parse error case

Parsing **bind** programs

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

parse

```
;; An AST is one of:
;; - …
```

Need to be more careful parsing

Valid Program?

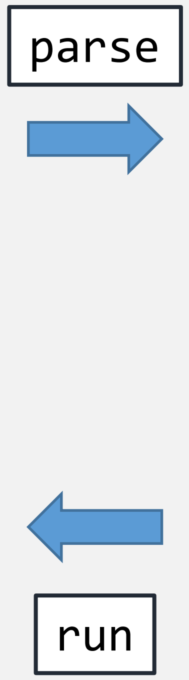
```
(check-exn exn:fail:syntax:cs450?
     (λ () (eval450 '(bind)) ))
```

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

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

Running **bind** programs

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



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


;; …
(struct vari [name])
(struct bind [x e body])
;; …
```

???

run, with accumulator

```
;; run: AST -> Result
;; Computes result of running a CS450 Lang program
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: remembers variable + values … currently in-scope
  (define (run/env p env)
    (match p
      …
      [(vari x) ...]
      [(bind x e body) ...]
      …   ))
 (run/env p  ???  ))
```

Parsing **bind** programs

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

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

```
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
                                  env)
```

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

```
      [(vari x) (env-lookup env x)]
      [(bind x e body) … (env-add env x (run/env e env)) …]
    …    ))
 (run/env p  ???  ))
```

Environment has **Results** (not **AST**)

How to convert **AST** to **Result**?

(From template!)

Add to environment

Be careful to get correct "**scoping**"
(x not visible in expression e,
so use unmodified input env)

Parsing **bind** programs

run must produce **Result**

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
                                   env)
```

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

```
     [(vari x) (env-lookup env x)]
     [(bind x e body) ??? (env-add env x (run/env e env)) …]
      …    ))
 (run/env p  ???  ))
```

Parsing **bind** programs

```
;; 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?

```
;; 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  ???  ))
```

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

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)
```

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

+ variable

Maps to internal “+” implementation (our “450+” function)

New kind of Result

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

For Program: +

Initial Environment

How do users call these functions???

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

```
(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: Examples

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

```
(+ 1 2)
```

Must be careful when parsing this

Function call case (must be last, why?)

```
(bind [x 1] 2)
```

(should not be parsed as function call)

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-call AST List<AST>)
;; …
(struct call [fn args])
```

"Running" Function Calls

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

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

“Running” Function Calls

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

TEMPLATE: extract pieces of compound data

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

"Running" Function Calls

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

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

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

TEMPLATE: recursive calls

List-processing function

“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
        …
     [(call fn args) (apply ???
                      (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
```

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

```
(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))
```

“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
          …
      [(call 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

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]))
```

Combine cases

Check correct number of arguments???

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

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

Interlude: Variable-arity functions in Racket

Programmer should not be constructing a list

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

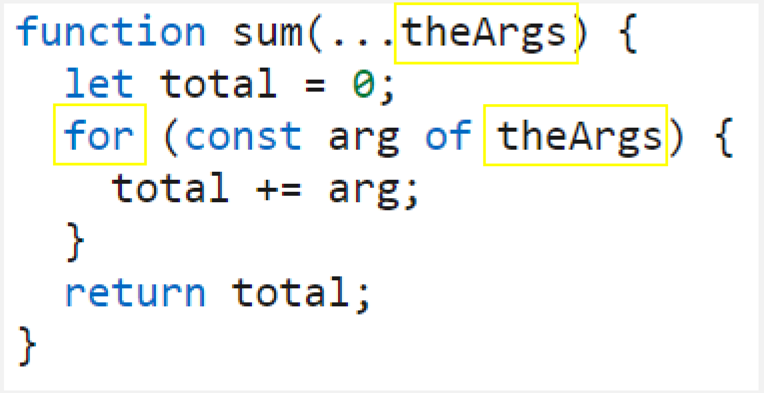
```
;; 450+: Result … -> Result
```

Inside the function, **args** is a **list of arguments**

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

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;
}
```

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)

This doesn't let users define their own functions!

Next Feature: **Lambdas?**

Function Application in CS450 Lang

What functions can be called?

```
(+ 1 2)
```

1. (Racket) functions in initial environment

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

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

“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

```
;; 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
```

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 ) )
   ? )
```

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])
```

Because this represents code!

Why can’t we use a Racket lambda?

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) ?? 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
```

What should be the “Result” of running a lm 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 ???
```

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

“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???

`lm-result` and `lm-ast` cannot be the same!!

(how can we **“remember”** the x)

In-class:
- try this in Racket (with **`lambda`** and **`let`**)
- find 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!)

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

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

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

We can’t!! need some other representation

```
;; 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 ???
```

“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))
```

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

Save the current **env**

Previously

"Running" Function Calls: Revisited

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

How do we actually run the 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))
```

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
      ...
     [(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"

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

TEMPLATE

```
;; 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) … ]))
```

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
```

In-class Coding 4/24: `lm` 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

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

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

Come up with some of <u>your own</u>!
(i.e., not my examples)
(can be error cases, both “syntax” and “result”)