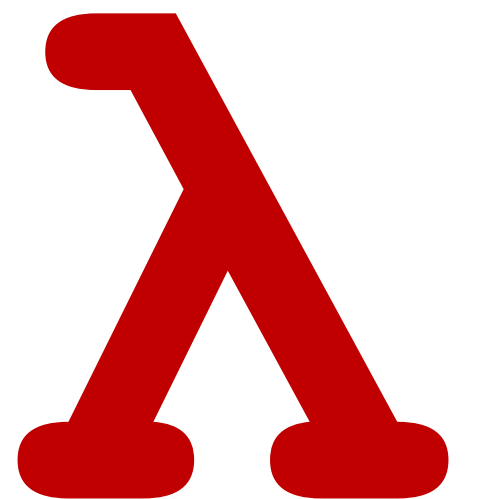


CSCI 275: Programming Abstractions

Lecture 24: MiniScheme F (Lambdas)

Fall 2024

Stephen Checkoway
Slides from Molly Q Feldman

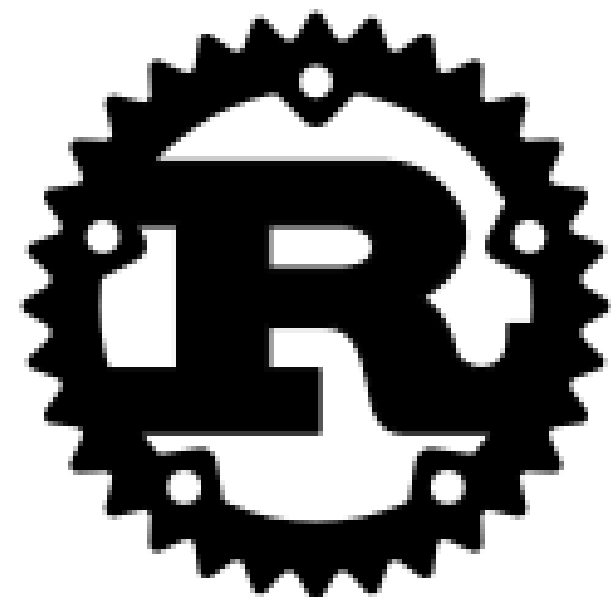


Functional Language of the Week: Rust

- 19th on the top 50 languages list
- I think the language that has transformed SE development the most in the last decade
 - Went public in 2010
 - Originally from Mozilla (creators of Firefox)
- What 241 is now taught in!

In the forward to the Rust Book: “the Rust programming language is fundamentally about *empowerment*: no matter what kind of code you are writing now, Rust empowers you to reach farther, to program with confidence in a wider variety of domains than you did before.”

Main use case? **Systems programming**



Functional Language of the Week: Rust

Core functional features in Rust are:

- Closures

```
fn add_one_v1 (x: u32) -> u32 { x + 1 }  
let add_one_v2 = |x: u32| -> u32 { x + 1 };  
let add_one_v3 = |x|           { x + 1 };  
let add_one_v4 = |x|           x + 1 ;
```

Same type inference
idea that we saw in
Kotlin & discussed with
Typed Racket

- Iterators

- They look at the performance and find that iterators are actually faster than loops: <https://doc.rust-lang.org/book/ch13-04-performance.html>



Note on MiniScheme Testing

- Make sure for parse, you do test the parse *tree* itself
 - Not just the type of the output
 - Not just the accessors

```
(app-exp (var-exp 'foo)
         (list (lit-exp 1) (lit-exp 4) (lit-exp 11)))
```

Test that the parse tree actually is
exactly this, not just that it produces
an app-exp

Reminder: Why MiniScheme?

Next 3 Lectures: MiniScheme Conclusion

Goal: go over key ideas behind the remaining parts of MiniScheme

What's left?

- lambdas: today
- set! and begin: Friday
- Recursion: Monday

It's quite a bit of **content**:
goal is get the main ideas
from the slides, then *review*
them when doing HW8

What's left in the MiniScheme Grammar?

$EXP \rightarrow$ number parse into `lit-exp`
| symbol parse into `var-exp`
| (if $EXP\ EXP\ EXP$) parse into `ite-exp`
| (let ($LET-BINDINGS$) EXP) parse into `let-exp`
| (letrec ($LET-BINDINGS$) EXP)
| (lambda ($PARAMS$) EXP) parse into `lambda-exp`
| (set! symbol EXP) parse into `set-exp`
| (begin EXP^*) parse into `begin-exp`
| ($EXP\ EXP^*$) parse into `app-exp`

$LET-BINDINGS \rightarrow LET-BINDING^*$

$LET-BINDING \rightarrow [\text{symbol } EXP]^*$

$PARAMS \rightarrow \text{symbol}^*$

Restatement of our Overall Goal

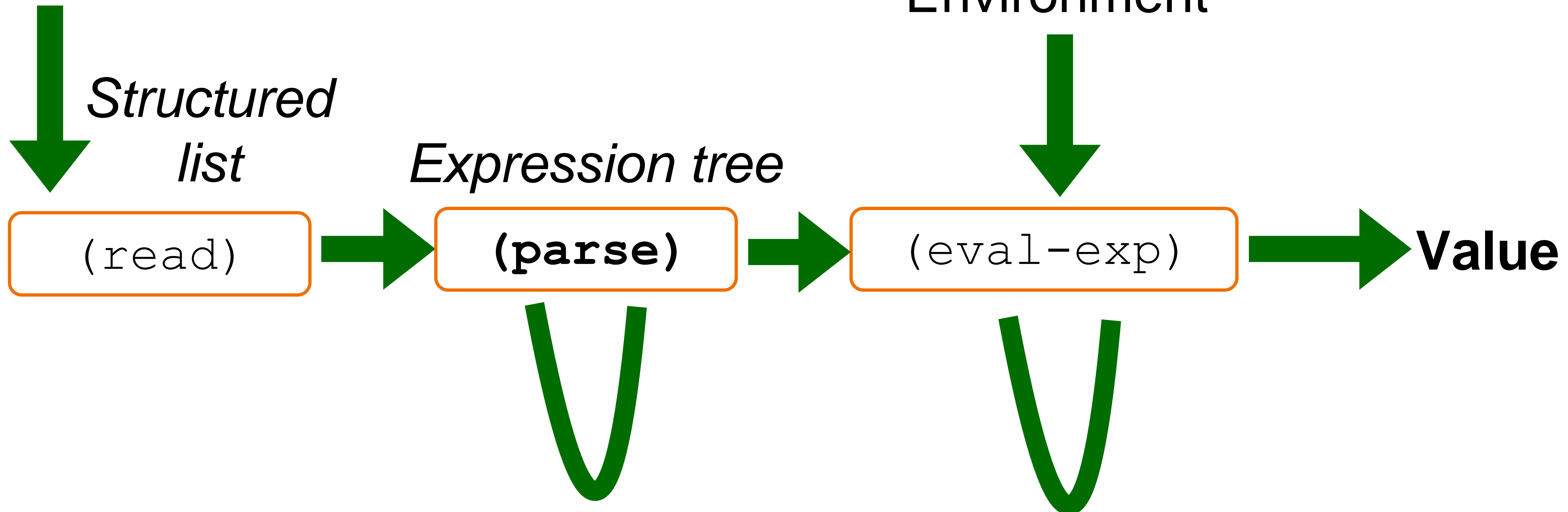
We have a language called MiniScheme, which we are building up piece-by-piece

We have a *formal* model of how it should work in a grammar, i.e. we know how to write it down

Our task: give it meaning – practically, determine values

MiniScheme Design

MiniScheme
expression as a string



Why do we need to do this?

`(if (gt? 2 3) (+ 2 3) 3)` could mean *anything*

We need to determine if it is:

- A valid MiniScheme expression - *parser*
- What value it would have - *interpreter*
 - Could be True, False, 5, 3, 'banana, etc.

Real World Example: CPython

If you've ever heard "Python is implemented in C", it *really is*

The backend of the Python interpreter is written in C, you can look at the source here:

<https://github.com/python/cpython>

Details of how parsing works for Python:

<https://github.com/python/cpython/blob/main/InternalDocs/compiler.md>

Back to MiniScheme Key Ideas

Review: How do we parse an application like `(+ 2 3)`?

A. `(app-exp + 2 3)`

B. `(app-exp + (2 3))`

C. `(app-exp (var-exp '+) (lit-exp 2) (lit-exp 3))`

D. `(app-exp (var-exp '+) (list (lit-exp 2) (lit-exp 3)))`

E. None of the above

At a higher level...

```
(app-exp (var-exp '+)  
        (list (lit-exp 2) (lit-exp 3)))
```

Applications are parsed into two parts

- The expression for the **procedure** part
- The list of **parsed arguments**

Reminder: Evaluating an `app-exp`

How do we evaluate the `app-exp` we get from

`(app-exp parsed-proc list-of-parsed-args)`?

In steps:

1. We evaluate the `parsed-proc` and the `list-of-parsed-args` in the current environment
2. Then we call `apply-proc` with the evaluated procedure and list of arguments

```
(define eval-exp ...  
  [(app-exp? tree)  
   (let ([proc (eval-exp (app-exp-proc tree) e)]  
         [args (map ... (app-exp-args tree))])  
     (apply-proc proc args))])
```

Now, let's add Lambdas

$EXP \rightarrow$ number parse into `lit-exp`
| symbol parse into `var-exp`
| (if EXP EXP EXP) parse into `ite-exp`
| (let ($LET-BINDINGS$) EXP) parse into `let-exp`
| (lambda ($PARAMS$) EXP) parse into `lambda-exp`
| (EXP EXP^*) parse into `app-exp`

$LET-BINDINGS \rightarrow LET-BINDING^*$

$LET-BINDING \rightarrow [\text{symbol } EXP]^*$

$PARAMS \rightarrow \text{symbol}^*$

Lambdas, in two stages

First, we want to think about parsing & evaluating *just* lambdas

```
MS> (lambda (x) x)
```

Second, we want to think about *applying* lambdas

```
MS> ((lambda (x) x) 45)
```

Parsing lambdas

Parse a lambda expression such as

`(lambda (x y z) body)` into a new `lambda-exp` data type

This needs

- The parameter list, e.g., `'(x y z)`
- the parsed `body`

Note that the **parameter list is not parsed**, it's just a list of symbols

Just like the symbols for binding
in `let-exp`

Evaluating for Lambdas

What should a `lambda-exp` evaluate to?

In other words, what is the result of evaluating something like `(lambda (x) (+ x y))`?

Reminder: closures

The expression of `(lambda parameters body..)` evaluates to a *closure* consisting of

- The parameter list (a list of identifiers)
- The body as un-evaluated expressions (often just one expression)
- The environment (the mapping of identifiers to values) **at the time the lambda expression is evaluated**

Closures!

We need a `(closure params body env)` data type!

```
(closure? obj)
```

```
(closure-params c)
```

```
(closure-body c)
```

```
(closure-env c)
```

closure data type

The `params` and the `body` come directly from the `lambda-exp`

The `env` is the current environment argument to `eval-exp`

Where should the new closure data type be defined? Why?

A. `parse.rkt`

B. `interp.rkt`

C. `closure.rkt`

D. `minischeme.rkt`

Summary of Handling `MS>` `(lambda (x) x)`

To parse a lambda

- Make a new `lambda-exp` object to hold parameters and body

To evaluate a lambda

- Make a new `closure` object to hold the parameters, body, and environment

Next *Calling* Lambda Expressions

```
MS> ((lambda (x) x) 45)
```

Nothing new is needed for **parsing calls** to lambda expressions; **why?**

```
(let ([f (lambda (x) (+ x y))])  
  (f (- a b)))
```


Parsing Calls MS> ((lambda (x) x) 45)

Answer: they are just application expressions!

```
(let ([f (lambda (x) (+ x y))])  
  (f (- a b)))
```

parses to:

```
(app-exp (var-exp 'f)  
  (list (app-exp (var-exp '-')  
    (list (var-exp 'a) (var-exp 'b))))))
```

Example: `((lambda (x y) (+ x y)) 3 5)`

Parse into an `(app-exp proc args)`

```
(app-exp (lambda-exp '(x y)
                    (app-exp (var-exp '+)
                              (list (var-exp 'x)
                                    (var-exp 'y))))
        (list (lit-exp 3)
              (lit-exp 5)))
```

For evaluating: we only handle primitives atm

Recall: All applications are evaluated by calling `apply-proc` with the evaluated procedure and the list of evaluated arguments

Here's what our `apply-proc` looks like after HW6

```
(define (apply-proc proc args)
  (cond [(prim-proc? proc)
        (apply-primitive-op
         (prim-proc-op proc) args)]
        [else (error "...)]))
```

Evaluating calls to closures

We need to add some code before the `else` to handle calls to closures!

```
(define (apply-proc proc args)
  (cond [(prim-proc? proc)
        (apply-primitive-op
         (prim-proc-op proc) args)]
        [(closure? proc) ...]
        [else (error "...)]))
```

Reminder: When to extend an environment?

There are only two places where an environment is extended:

A. Let expressions

B. Procedure calls

How do we evaluate the closure?

In general in Racket, given a closure and some arguments, how do we evaluate calling the closure?

Steps

- Extend the closure's environment with bindings from the closure's parameters to argument values
- Evaluate the body of the closure in this extended environment

If you find yourself wanting to pass the environment from `eval-exp` to `apply-proc`, there is something wrong; you don't need to do that

The *Closure's* Environment

When we apply the closure to argument expressions

- we evaluate the arguments in the current environment
- extend the **closure's** environment with bindings of parameters to argument values
- evaluate the closure's body in the extended environment

MiniScheme (and Racket) are *lexically scoped* languages –we'll talk more about this next week!

Evaluating `((lambda (x y) (+ x y)) 3 5)`

```
(app-exp (lambda-exp '(x y)
                    (app-exp (var-exp '+)
                              (list (var-exp 'x)
                                    (var-exp 'y))))
        (list (lit-exp 3) (lit-exp 5)))
```

This is evaluated by calling `apply-proc` with the evaluated procedure and evaluated arguments

Evaluating the **procedure** part of the `app-exp` gives

```
(closure '(x y)
         (app-exp (var-exp '+)
                   (list (var-exp 'x) (var-exp 'y))))
e)
```

Evaluating the **arguments** gives `'(3 5)`

Evaluating `((lambda (x y) (+ x y)) 3 5)`

`apply-proc` will evaluate the closure

```
(closure ' (x y)
         (app-exp (var-exp '+)
                  (list (var-exp 'x) (var-exp 'y))))
e)
```

by calling `eval-exp` on the **body** in the environment

```
e [x ↦ 3, y ↦ 5]
```

Since the body is an `app-exp`, it'll evaluate `(var-exp '+)` to get `(prim-proc '+)` and the arguments to get `' (3 5)`

Another Example: Parsing

What is the result of parsing this?

```
(let ([f (lambda (x) (* 2 x))])  
      (f 6))
```

Another Example: Parsing

What is the result of parsing this?

```
(let ([f (lambda (x) (* 2 x))])  
      (f 6))
```

Result:

```
(let-exp ' (f)  
        (list (lambda-exp  
                ' (x)  
                (app-exp (var-exp '*)  
                          (list (lit-exp 2)  
                                (var-exp 'x))))))  
        (app-exp (var-exp 'f)  
                  (list (lit-exp 6))))
```

Reminder: Evaluating `let` expressions

1. Evaluate each of the binding expressions in the `let-exp`
2. Bind the symbols to these values by extending the current environment
3. Evaluate the body of the `let` expression using the extended environment

Another Example: Evaluating

```
(let-exp ' (f)
  (list (lambda-exp
        ' (x)
        (app-exp (var-exp '*)
                  (list (lit-exp 2)
                        (var-exp 'x))))))
(app-exp (var-exp 'f)
  (list (lit-exp 6))))
```

Only one binding in
the `let`

Evaluate the `let-exp` by extending the current environment e with f bound to the closure we get by evaluating the `lambda-exp` in environment e

Another Example: Evaluating

With f bound to

```
(closure ' (x)
          (app-exp (var-exp '*)
                   (list (lit-exp 2) (var-exp 'x)))
          e)
```

we next evaluate the body of the let

```
(app-exp (var-exp 'f) (list (lit-exp 6)))
```

This will evaluate `(var-exp 'f)`—getting the closure above—
and evaluate the arguments getting `'(6)`

`apply-proc` will call `eval-exp` on the **body of the closure** and
the environment $e[x \mapsto 6]$

This is another application expression, and the process continues!